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 a sophisticated datatype into a basic datatype and a predicate on the basic datatype. More specifically, we assume that the sophisticated datatype <u>and the basic datatype</u> are known and their descriptions (**??**) are straightforwardly related by an **ornament** (Section 3.2), and derive from the ornament an externalist predicate and an indexed family of conversion isomorphisms. 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 task can be broken into two parts:

- coordination of relevant conversion isomorphisms for upgrading basic operations satisfying suitable properties to have more sophisticated (function) types, and
- manufacture of conversion isomorphisms between the datatypes involved.

Refinements (Section 3.1), which axiomatise conversion isomorphisms between internalist and externalist datatypes, are the abstraction we introduce for bridging the two parts. The first part is then formalised with **upgrades** (Section 3.1.2) which use refinements as their components, and the second part is done by translating ornaments to refinements (Section 3.3). To actually harvest externalist composability, we need conversion isomorphisms in which the externalist predicates involved are pointwise conjunctions (as in the case of externalist ordered vectors in **??**). Such conversion isomorphisms come from **parallel composition** of ornaments (Section 3.2.3), which not only gives rise to pointwise conjunctive predicates on the externalist side (Section 3.3.2) but also produces composite datatypes on the internalist side (e.g., the internalist datatype of ordered vectors incorporating both ordering and length information). The above framework of ornaments, refinements, and upgrades are illustrated with several examples in Section 3.4, followed by some discussion (including related work) in Section 3.5.

3.1 Refinements

We first abstract away from the detail of the construction of conversion isomorphisms and simply axiomatise their existence as **refinements** from basic types to more sophisticated types. There are two versions of refinements:

- the non-indexed version between individual types (Section 3.1.1), and
- the indexed version between two families of types — called **refinement families** (Section 3.1.3) — which collect refinements between specified pairs of individual types in the two families.

Section 3.1.2 then explains how refinements between individual types can be coordinated to perform function upgrading, and the actual construction of a class of refinement families is described in Section 3.3 after the introduction of ornaments (Section 3.2).

3.1.1 Refinements between individual types

A **refinement** from a basic type A to a more informative type B is a **promotion predicate** $P : A \to \mathsf{Set}$ and a **conversion isomorphism** $i : B \cong \Sigma\, A\, P$. As an AGDA record datatype:

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Refinement}\ (A\ B : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad P : A \to \mathsf{Set} \\
\qquad i\ : B \cong \Sigma\ A\ P \\
\quad \mathit{forget} : B \to A \quad \text{-- explained after the two examples below} \\
\quad \mathit{forget} = \mathit{outl} \circ \mathsf{Iso}.\mathit{to}\ i
\end{array}
$$

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

Example (*refinement from lists to ordered lists*)**.** Consider the internalist datatype of ordered lists (indexed by a lower bound; the type Val and associated operations are postulated in **??**):

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{OrdList} : \mathit{Val} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{OrdList}\ b \ni \mathsf{nil} \\
\qquad\qquad\quad \mid \mathsf{cons}\ (x : \mathit{Val})\ (\mathit{leq} : b \leqslant x)\ (\mathit{xs} : \mathsf{OrdList}\ x)
\end{array}
$$

Fixing $b : \mathit{Val}$, there is a refinement from $\mathsf{List}\ \mathit{Val}$ to $\mathsf{OrdList}\ b$ whose promotion predicate is $\mathsf{Ordered}\ b$, since we have an isomorphism of type

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

which, from left to right, decomposes an ordered list into the underlying list and a proof that the underlying list is ordered (and bounded below). Conversely, a list satisfying $\mathsf{Ordered}\ b$ can be promoted to an ordered list of type $\mathsf{OrdList}\ b$ by the right-to-left direction of the isomorphism. □

Example (*refinement from natural numbers to lists*)**.** Let $A : \mathsf{Set}$ (which we will directly refer to in subsequent text and code as if it is a local module parameter). We have a refinement from Nat to $\mathsf{List}\ A$

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

for which $\mathsf{Vec}\ A$ 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 $n\ :\ \mathsf{Nat}$ is an incomplete list — the list elements are missing from the successor nodes of n. To promote n to a $\mathsf{List}\ A$, we need to supply a vector of type $\mathsf{Vec}\ A\ n$, 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 inhabitants 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. □

Given a refinement r, we denote the forgetful computation of underlying values — i.e., $\mathit{outl} \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.\mathit{i}\ r)$ — as $\mathsf{Refinement}.\mathit{forget}\ r$. (This is done by defining an extra projection function *forget* in the record definition of Refinement.) The forgetful function is actually the core of a refinement, as justified by the following facts:

- The forgetful function determines a refinement extensionally — if the forgetful functions of two refinements are extensionally equal, then their promotion predicates are pointwise isomorphic:

$$\begin{aligned}\mathit{forget\text{-}iso}\ :\ & \{A\ B\ :\ \mathsf{Set}\}\ (r\ s\ :\ \mathsf{Refinement}\ A\ B) \rightarrow \\ & (\mathsf{Refinement}.\mathit{forget}\ r\ \doteq\ \mathsf{Refinement}.\mathit{forget}\ s) \rightarrow \\ & (a\ :\ A) \rightarrow \mathsf{Refinement}.P\ r\ a\ \cong\ \mathsf{Refinement}.P\ s\ a\end{aligned}$$

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

$$\mathit{canonRef}\ :\ \{A\ B\ :\ \mathsf{Set}\} \rightarrow (B \rightarrow A) \rightarrow \mathsf{Refinement}\ A\ B$$

$$
\begin{array}{l}
\mathit{canonRef}\ \{A\}\ \{B\}\ f\ =\ \textbf{record} \\
\quad \{\ P\ =\ \lambda\, a \mapsto \Sigma[\, b : B\,]\ f\ b \equiv a \\
\quad ;\ i\ \ =\ \textbf{record} \\
\qquad\qquad \{\ \mathit{to}\ \ \ =\ f \vartriangle (\mathit{id} \vartriangle (\lambda\, b \mapsto \mathsf{refl}))\quad \text{-- } (g \vartriangle h)\ x\ =\ (g\ x\ , h\ x) \\
\qquad\qquad ;\ \mathit{from}\ =\ \mathit{outl} \circ \mathit{outr} \\
\qquad\qquad ;\ \boxed{\text{proofs of laws}}\ \}\ \}\quad \text{-- proofs of inverse properties omitted}
\end{array}
$$

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

- For any refinement $r : \mathsf{Refinement}\ A\ B$, its forgetful function is definitionally 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:

$$
\begin{array}{l}
\mathit{coherence}\ : \\
\quad \{A\ B\ :\ \mathsf{Set}\}\ (r\ :\ \mathsf{Refinement}\ A\ B) \to \\
\quad (a\ :\ A) \to \mathsf{Refinement}.P\ r\ a \cong \Sigma[\, b : B\,]\ \mathsf{Refinement}.\mathit{forget}\ r\ b \equiv a \\
\mathit{coherence}\ r\ a\ =\ \mathit{forget\text{-}iso}\ r\ (\mathit{canonRef}\ (\mathsf{Refinement}.\mathit{forget}\ r))\ (\lambda\, b \mapsto \mathsf{refl})
\end{array}
$$

  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 to 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 $\mathsf{Ordered}$ as its promotion predicate. A promotion proof of type $\mathsf{Ordered}\ b\ xs$ for the list xs consists of only the inequality proofs necessary for ensuring that xs 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:

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Refinement}'\ (A\ B\ :\ \mathsf{Set})\ :\ \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad \begin{array}{lcl}
P & : & A \to \mathsf{Set} \\
\mathit{forget} & : & B \to A \\
p & : & (a\ :\ A) \to P\ a \cong \Sigma[\,b : B\,]\ \mathit{forget}\ b \equiv a
\end{array}
\end{array}
$$

We say that $a\ :\ A$ and $b\ :\ B$ are **in coherence** when $\mathit{forget}\ b\ \equiv\ a$, i.e., when a underlies b. The two definitions of refinements are equivalent. Of particular importance is the direction from $\mathsf{Refinement}$ to $\mathsf{Refinement}'$:

$$
\begin{array}{l}
\mathit{toRefinement}'\ :\ \{A\ B\ :\ \mathsf{Set}\} \to \mathsf{Refinement}\ A\ B \to \mathsf{Refinement}'\ A\ B \\
\mathit{toRefinement}'\ r\ =\ \textbf{record}\ \begin{array}[t]{lcl}
\{\ P & = & \mathsf{Refinement}.P\ r \\
;\ \mathit{forget} & = & \mathsf{Refinement}.\mathit{forget}\ r \\
;\ p & = & \mathit{coherence}\ r\ \}
\end{array}
\end{array}
$$

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, can be more easily generalised 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 use 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 $\mathsf{List}\ \mathsf{Nat}$ are related by a refinement: if such a refinement existed, we would be able to extract from any function $f : \mathsf{List\ Nat} \to \mathsf{List\ Nat}$ 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 $\mathsf{List\ Nat} \to \mathsf{List\ Nat}$ 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 doubling a natural number

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

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

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

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

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

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

$$
\begin{aligned}
& \mathit{duplicate} : \mathsf{List}\ A \to \mathsf{List}\ A \\
& \mathit{duplicate} = \mathsf{Iso}.\mathit{from}\ i \circ (\mathit{double} * \mathit{duplicate}'\ \_) \circ \mathsf{Iso}.\mathit{to}\ i
\end{aligned}
$$

That is, we decompose the input list into the underlying natural number (i.e., its length) and a vector of elements, process the two parts separately with *double* and *duplicate*′, and finally combine the results back to a list. (This is what we did for insert_V in **??**.) The relationship between the promoted function *duplicate* and the underlying function *double* is characterised by the **coherence property**

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

or as a commutative diagram:

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

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*.

We thus define **upgrades** to capture the promoting direction and the coherence property abstractly. An upgrade from $A : \mathsf{Set}$ to $B : \mathsf{Set}$ is

- a promotion predicate $P : A \to \mathsf{Set}$,
- a coherence property $C : A \to B \to \mathsf{Set}$ relating inhabitants of the basic type A and inhabitants of the more informative type B,
- an upgrading (promoting) operation $u : (a : A) \to P\ a \to B$, and
- a coherence proof $c : (a : A)\ (p : P\ a) \to C\ a\ (u\ a\ p)$ saying that the result of promoting $a : A$ must be in coherence with a.

As an Agda record datatype:

$$\begin{array}{l}
\textbf{record}\ \mathsf{Upgrade}\ (A\ B : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\quad\quad P : A \to \mathsf{Set} \\
\quad\quad C : A \to B \to \mathsf{Set} \\
\quad\quad u\ : (a : A) \to P\ a \to B \\
\quad\quad c\ : (a : A)\ (p : P\ a) \to C\ a\ (u\ a\ p)
\end{array}$$

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:

$$\begin{aligned}
&\mathit{toUpgrade}' \;:\; \{A\; B \;:\; \mathsf{Set}\} \to \mathsf{Refinement}'\; A\; B \to \mathsf{Upgrade}\; A\; B\\
&\mathit{toUpgrade}'\; r \;=\; \textbf{record}\; \{\; P \;=\; \mathsf{Refinement}'.P\; r\\
&\qquad\qquad\qquad\qquad\qquad ;\; C \;=\; \lambda\, a\; b \mapsto \mathsf{Refinement}'.\mathit{forget}\; r\; b \;\equiv\; a\\
&\qquad\qquad\qquad\qquad\qquad ;\; u \;=\; \lambda\, a \mapsto \mathit{outl} \circ \mathsf{Iso}.\mathit{to}\; (\mathsf{Refinement}'.p\; r\; a)\\
&\qquad\qquad\qquad\qquad\qquad ;\; c \;=\; \lambda\, a \mapsto \mathit{outr} \circ \mathsf{Iso}.\mathit{to}\; (\mathsf{Refinement}'.p\; r\; a)\; \}
\end{aligned}$$

and consequently any refinement gives rise to an upgrade:

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

Composition of upgrades

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

$$\begin{aligned}
&_\rightharpoonup_ \;:\; \{A\; A'\; B\; B' \;:\; \mathsf{Set}\} \to\\
&\qquad\quad \mathsf{Refinement}\; A\; A' \to \mathsf{Upgrade}\; B\; B' \to \mathsf{Upgrade}\; (A \to B)\; (A' \to B')
\end{aligned}$$

Note that there should be a refinement between the source types A and A', 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 to be able to decompose the source type A'.

Let $r : \mathsf{Refinement}\ A\ A'$ and $s : \mathsf{Upgrade}\ B\ B'$. The upgrading operation takes a function $f : A \to B$ and combines it with a promotion proof to get a function $f' : A' \to B'$, which should transform underlying values in a way that is in coherence with f. That is, as f' takes $a' : A'$ to $f'\ a' : B'$ at the more informative level, correspondingly at the underlying level, the value underlying a', i.e., $\mathsf{Refinement}.\mathit{forget}\ r\ a' : A$, should be taken by f to a value in coherence with $f'\ a'$. We thus define the statement "f' is in coherence with f" as

$$(a : A)\ (a' : A') \to \mathsf{Refinement}.\mathit{forget}\ r\ a' \equiv a \to \mathsf{Upgrade}.C\ s\ (f\ a)\ (f'\ a')$$

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 $a : A$ how a promotion proof for a is transformed to a promotion proof for $f\ a$. The type of promotion proofs for f is thus

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

Having determined the coherence property and the promotion predicate, it is then easy to construct the upgrading operation and the coherence proof. In particular, the upgrading operation

- breaks an input $a' : A'$ into its underlying value $a = \mathsf{Refinement}.\mathit{forget}\ r\ a' : A$ and a promotion proof for a,
- computes a promotion proof q for $f\ a : B$ using the given promotion proof for f, and
- promotes $f\ a$ to an inhabitant of type B' using q.

which is an abstract version of what we did in the *double–duplicate* example. The complete definition of $\_\rightharpoonup\_$ is

$$
\begin{aligned}
&\_\rightharpoonup\_ : \{A\ A'\ B\ B' : \mathsf{Set}\} \to \\
&\qquad \mathsf{Refinement}\ A\ A' \to \mathsf{Upgrade}\ B\ B' \to \mathsf{Upgrade}\ (A \to B)\ (A' \to B') \\
&r \rightharpoonup s = \mathbf{record} \\
&\quad \{\ P = \lambda f \mapsto (a : A) \to \mathsf{Refinement}.P\ r\ a \to \mathsf{Upgrade}.P\ s\ (f\ a) \\
&\quad ;\ C = \lambda f\ f' \mapsto (a : A)\ (a' : A') \to \\
&\qquad\qquad\qquad \mathsf{Refinement}.\mathit{forget}\ r\ a' \equiv a \to \mathsf{Upgrade}.C\ s\ (f\ a)\ (f'\ a')
\end{aligned}
$$

$$
\begin{aligned}
&;\ u \ = \ \lambda f\ h \mapsto \mathsf{Upgrade}.u\ s\ \_ \circ \mathit{uncurry}\ h \circ \mathsf{Iso}.to\ (\mathsf{Refinement}.i\ r) \\
&;\ c \ = \ \lambda\ \{\ f\ h\ .\_\ a'\ \mathsf{refl} \mapsto \mathbf{let}\ (a\ ,p) \ = \ \mathsf{Iso}.to\ (\mathsf{Refinement}.i\ r)\ a' \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathsf{Upgrade}.c\ s\ (f\ a)\ (h\ a\ p)\ \}\ \}
\end{aligned}
$$

Example (*upgrade from* $\mathsf{Nat} \to \mathsf{Nat}$ *to* $\mathsf{List}\ A \to \mathsf{List}\ A$)**.** Using the $\_\rightharpoonup\_$ combinator on the refinement

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

and the upgrade derived from r by *toUpgrade*, 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 *duplicate*′, and the type $\mathsf{Upgrade}.C\ u\ \mathit{double}\ \mathit{duplicate}$ is exactly the coherence property satisfied by *double* and *duplicate*. □

A further example on upgrades (about insertion into a binomial heap) is given in Section 3.4.2.

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 particular collection of types in Y, the underlying value 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** below, the index-level refinement degenerates into just the forgetful function.

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

The inverse image type $\_^{-1}\_$ is defined by

$$\textbf{data}\ \_^{-1}\_\ (e : J \to I)\ (i : I) : \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{ok} : (j : J) \to e^{\,-1}\ (e\ j)$$

That is, $e^{\,-1}\ i$ is isomorphic to $\Sigma[\,j : J\,]\ \ e\ j \equiv i$, the subset of J mapped to i by e. An underlying J-value is extracted by

$$\mathit{und} : \{I\,J : \mathsf{Set}\}\ \{e : J \to I\}\ \{i : I\} \to e^{\,-1}\ i \to J$$
$$\mathit{und}\ (\mathsf{ok}\ j)\ =\ j$$

Introducing this type will offer some slight notational advantage when, e.g., writing ornamental descriptions (Section 3.2.2). We also define an alternative name $\mathsf{InvImage}\ =\ \_^{-1}\_$ to make partial application look better.

Example (*refinement family from ordered lists to ordered vectors*)**.** The datatype $\mathsf{OrdList} : \mathit{Val} \to \mathsf{Set}$ 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 the datatype of ordered vectors $\mathsf{OrdVec} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set}$:

$$\textbf{indexfirst data}\ \mathsf{OrdVec} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{OrdVec}\ b\ \mathsf{zero} \quad\ni\ \mathsf{nil}$$
$$\quad \mathsf{OrdVec}\ b\ (\mathsf{suc}\ n)\ \ni\ \mathsf{cons}\ (x : \mathit{Val})\ (\mathit{leq} : b \leqslant x)\ (\mathit{xs} : \mathsf{OrdVec}\ x\ n)$$

This further classification is captured as a refinement family of type

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

which consists of refinements from $\mathsf{OrdList}\ b$ to $\mathsf{OrdVec}\ b\ n$ for all $b : \mathit{Val}$ and $n : \mathsf{Nat}$. $\square$

Refinement families are the vehicle we use to express conversion relationship between inductive families. For now, however, they have to be prepared manually, which requires considerable effort. Also, when it comes to acquiring externalist composability for internalist datatypes, we need to be able to compose refinements such that the promotion predicate of the resulting refinement is the pointwise conjunction of existing promotion predicates, so we get conversion isomorphisms of the right form. For example, we should be able to compose the two refinements from lists to ordered lists and vectors to get a refinement from lists to ordered vectors whose promotion predicate is the pointwise conjunction of the promotion predicates of the two refinements. This is easy for the externalist side of the refinement, but for the internalist side, we need to derive the datatype of ordered vectors from the datatypes of ordered lists and vectors, which is not possible unless we can tap more deeply into the structure of datatypes and manipulate such structure — that is, we need to do **datatype-generic programming** (**??**). Hence enter ornaments. With ornaments, we can express intensional relationship between datatype <u>declarations</u>, which can be exploited for deriving composite datatypes like ordered vectors. This intensional relationship is easy to establish and induces refinement families (Section 3.3), so the difficulty of preparing refinement families is also dramatically reduced.

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 $\mathbb{F}\ \mathit{NatD}$:

$$
\begin{array}{lcll}
\mathit{erase} : \{A : \mathsf{Set}\}\ \{X : \top \to \mathsf{Set}\} \to \mathbb{F}\ (\mathit{ListD}\ A)\ X \rightrightarrows \mathbb{F}\ \mathit{NatD}\ X & & & \\
\mathit{erase}\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) & = & \text{‘}\mathsf{nil}\ ,\ \blacksquare & \text{-- ‘nil copied} \\
\mathit{erase}\ (\text{‘}\mathsf{cons}\ ,\ a\ ,\ x\ ,\ \blacksquare) & = & \text{‘}\mathsf{cons}\ ,\ x\ ,\ \blacksquare & \text{-- ‘cons copied, } a \text{ discarded,} \\
 & & & \text{-- and } x \text{ retained}
\end{array}
$$

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

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

(Implicit arguments can be explicitly supplied in curly braces.) 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 (e.g., by parallel composition in Section 3.2.3), 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

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

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

$$\textbf{data}\ \mathsf{ROrn}\ \{I\ J : \mathsf{Set}\}\ (e : J \to I) : \mathsf{RDesc}\ I \to \mathsf{RDesc}\ J \to \mathsf{Set}_1$$

$$\begin{aligned}
&\mathit{erase} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{RDesc}\ I\}\ \{E : \mathsf{RDesc}\ J\} \to \\
&\qquad \mathsf{ROrn}\ e\ D\ E \to \{X : I \to \mathsf{Set}\} \to [\![\ E\ ]\!]\ (X \circ e) \to [\![\ D\ ]\!]\ X
\end{aligned}$$

The datatype Orn is parametrised by an erasure function $e : J \to I$ on the index sets and relates a basic description $D : \mathsf{Desc}\ I$ with a more informative description $E : \mathsf{Desc}\ J$. (We sometimes refer to $\mu\ E$ (e.g., lists) as an **ornamentation** of $\mu\ D$ (e.g., natural numbers).) As a consequence, 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:

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

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

$$\begin{aligned}
&\mathit{forget}\ O\ :\ \mu\ E \rightrightarrows (\mu\ D \circ e)\\
&\mathit{forget}\ O\ =\ \mathit{fold}\ (\mathit{ornAlg}\ O)
\end{aligned}$$

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 (whose indexing style corresponds to that of refinement families).

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

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

$$\begin{array}{lll}
\sigma & : & (S : \mathsf{Set})\ \{D : S \to \mathsf{RDesc}\ I\}\ \{E : S \to \mathsf{RDesc}\ J\} \\
& & (O : (s : S) \to \mathsf{ROrn}\ e\ (D\ s)\ (E\ s)) \to \mathsf{ROrn}\ e\ (\sigma\ S\ D)\ (\sigma\ S\ E) \\
\Delta & : & (T : \mathsf{Set})\ \{D : \mathsf{RDesc}\ I\}\ \{E : T \to \mathsf{RDesc}\ J\} \\
& & (O : (t : T) \to \mathsf{ROrn}\ e\ D\ (E\ t)) \to \mathsf{ROrn}\ e\ D\ (\sigma\ T\ E) \\
\nabla & : & \{S : \mathsf{Set}\}\ (s : S)\ \{D : S \to \mathsf{RDesc}\ I\}\ \{E : \mathsf{RDesc}\ J\} \\
& & (O : \mathsf{ROrn}\ e\ (D\ s)\ E) \to \mathsf{ROrn}\ e\ (\sigma\ S\ D)\ E
\end{array}$$

$$\begin{array}{l}
\mathit{erase} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{RDesc}\ I\}\ \{E : \mathsf{RDesc}\ J\} \to \\
\qquad \mathsf{ROrn}\ e\ D\ E \to \{X : I \to \mathsf{Set}\} \to [\![\ E\ ]\!]\ (X \circ e) \to [\![\ D\ ]\!]\ X \\
\mathit{erase}\ (\mathsf{v}\ [\,]\qquad\quad)\ \blacksquare \qquad\quad = \blacksquare \\
\mathit{erase}\ (\mathsf{v}\ (\mathsf{refl} :: \mathit{eqs}))\ (x , \mathit{xs}) = x , \mathit{erase}\ (\mathsf{v}\ \mathit{eqs})\ \mathit{xs} \quad \text{-- } x \text{ retained} \\
\mathit{erase}\ (\sigma\ S\ O)\qquad\quad (s , \mathit{xs}) = s , \mathit{erase}\ (O\ s)\quad \mathit{xs} \quad \text{-- } s \text{ copied} \\
\mathit{erase}\ (\Delta\ T\ O)\qquad\quad (t , \mathit{xs}) = \quad \mathit{erase}\ (O\ t)\quad \mathit{xs} \quad \text{-- } t \text{ discarded} \\
\mathit{erase}\ (\nabla\ s\ O)\qquad\qquad\quad \mathit{xs} = s , \mathit{erase}\ O\qquad \mathit{xs} \quad \text{-- } s \text{ inserted}
\end{array}$$

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 $\mathsf{v}\ js$ refines another response description $\mathsf{v}\ is$, i.e., when $[\![\ \mathsf{v}\ js\ ]\!]\ (X \circ e)$ can be transformed into $[\![\ \mathsf{v}\ is\ ]\!]\ X$. The source type $[\![\ \mathsf{v}\ js\ ]\!]\ (X \circ e)$ expands to a product of types of the form $X\ (e\ j)$ for some $j : J$ and the target type $[\![\ \mathsf{v}\ is\ ]\!]\ X$ 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 $\mathit{map}\ e\ js \equiv is$ (where *map* is the usual functorial map on lists). To make it easier to analyse a proof of $\mathit{map}\ e\ js \equiv is$ in the v case of *erase*, we instead define the proposition inductively as $\mathbb{E}\ e\ js\ is$, where the datatype $\mathbb{E}$ is defined by

$$\begin{array}{l}
\mathbf{data}\ \mathbb{E}\ \{I\ J : \mathsf{Set}\}\ (e : J \to I) : \mathsf{List}\ J \to \mathsf{List}\ I \to \mathsf{Set}\ \mathbf{where} \\
\quad [\,] \quad\ : \mathbb{E}\ e\ [\,]\ [\,] \\
\quad _{::}_ : \{j : J\}\ \{i : I\}\ (eq : e\ j \equiv i) \to \\
\qquad\qquad \{js : \mathsf{List}\ J\}\ \{is : \mathsf{List}\ I\}\ (eqs : \mathbb{E}\ e\ js\ is) \to \mathbb{E}\ e\ (j :: js)\ (i :: is)
\end{array}$$

- The σ case of ROrn states that $\sigma\ S\ E$ refines $\sigma\ S\ D$, 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 $A : \mathsf{Set}$, there is an ornament from the description *NatD* of natural numbers to the description *ListD A* of lists:

$$
\begin{array}{l}
\mathit{NatD\text{-}ListD}\ A : \mathsf{Orn}\ !\ \mathit{NatD}\ (\mathit{ListD}\ A)\\
\mathit{NatD\text{-}ListD}\ A\ (\mathsf{ok}\ \blacksquare) \;=\; \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{'}\mathsf{nil} \quad\mapsto\ \mathsf{v}\ []\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ \text{'}\mathsf{cons} \ \mapsto\ \Delta[\,\_ : A\,]\ \ \mathsf{v}\ (\mathsf{refl} :: [])\ \}
\end{array}
$$

where the erasure function '!' is $\lambda\ \_ \mapsto \blacksquare$. 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

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

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

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

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

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

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

computes the underlying list of a vector. $\square$

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

```
data ROrnDesc {I : Set} (J : Set) (e : J → I) : RDesc I → Set₁ where
  v : {is : List I} (js : ℙ is (InvImage e)) → ROrnDesc J e (v 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 (v {is} js) = v (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 (v js)     = v (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 3.1 Definitions for ornamental descriptions.

(e.g., *ListD A*), intending it to be a more refined version of an existing one (e.g., *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 related definitions are shown in Figure 3.1 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 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*). We can define ordered lists by an ornamental description, using the description of lists as the template:

$$\begin{aligned}
&\mathit{OrdListOD} \;:\; \mathsf{OrnDesc}\; \mathit{Val}\; !\; (\mathit{ListD}\; \mathit{Val})\\
&\mathit{OrdListOD}\; (\mathsf{ok}\; b) \;=\\
&\quad \sigma\; \mathsf{ListTag}\; \lambda\; \{\; \text{‘}\mathsf{nil} \quad\mapsto\; \mathsf{v}\; \blacksquare\\
&\qquad\qquad\qquad ;\; \text{‘}\mathsf{cons} \;\mapsto\; \sigma[\,x : \mathit{Val}\,]\;\; \Delta[\,\mathit{leq} : b \leqslant x\,]\;\; \mathsf{v}\; (x\;,\; \blacksquare)\;\}
\end{aligned}$$

If we read *OrdListOD* 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. We write

$$\lfloor \mathit{OrdListOD} \rfloor \;:\; \mathsf{Desc}\ \mathit{Val}$$

to decode *OrdListOD* to an ordinary description of ordered lists (in particular, turning the Δ into a σ) and

$$\lceil \mathit{OrdListOD} \rceil \;:\; \mathsf{Orn}\ !\ (\mathit{ListD}\ \mathit{Val})\ \lfloor \mathit{OrdListOD} \rfloor$$

to get an ornament from lists to ordered lists. □

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

$$\begin{array}{l} \textbf{indexfirst data}\ \mathsf{ListS}\ A\ :\ \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{ListS}\ A\ []\qquad\qquad \ni\ \mathsf{nil} \\ \quad \mathsf{ListS}\ A\ (a :: as)\ \ni\ \mathsf{cons}\ (s\ :\ \mathsf{ListS}\ A\ as) \end{array}$$

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

$$\begin{array}{l} \mathit{ListSOD}\ :\ (A\ :\ \mathsf{Set}) \to \mathsf{OrnDesc}\ (\mathsf{List}\ A)\ !\ (\mathit{ListD}\ A) \\ \mathit{ListSOD}\ A\ (\mathsf{ok}\ []\qquad\quad)\ =\ \nabla[\,\text{'}\mathsf{nil}\,]\ \mathsf{v}\ \blacksquare \\ \mathit{ListSOD}\ A\ (\mathsf{ok}\ (a :: as))\ =\ \nabla[\,\text{'}\mathsf{cons}\,]\ \nabla[\,a\,]\ \mathsf{v}\ (\mathsf{ok}\ as\ ,\ \blacksquare) \end{array}$$

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 general, we can define a parametrised ornamental description

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

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

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

$$\begin{array}{l} \mathit{singletonOD} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) \to \mathsf{OrnDesc}\ (\Sigma\ I\ (\mu\ D))\ \mathit{outl}\ D \\ \mathit{singletonOD}\ D\ (\mathsf{ok}\ (i\ , \mathsf{con}\ \mathit{ds})) = \mathit{erode}\ (D\ i)\ \mathit{ds} \end{array}$$

where

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

Note that *erode* deletes all fields (i.e., horizontal content), 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 (*index-first universes*)**.** The datatype of response ornamental descriptions is a good candidate for receiving an index-first reformulation. Since the structure of a response ornamental description is guided by the template response description, ROrnDesc is much more clearly presented in the index-first style:

$$\begin{array}{l} \textbf{indexfirst data}\ \mathsf{ROrnDesc}\ \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I) : \mathsf{RDesc}\ I \to \mathsf{Set}_1 \\ \quad \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{ROrnDesc}\ J\ e\ (\mathsf{v}\ \mathit{is}) & \ni & \mathsf{v}\ (\mathit{js} : \mathbb{P}\ \mathit{is}\ (\mathsf{InvImage}\ e)) \\ \mathsf{ROrnDesc}\ J\ e\ (\sigma\ S\ D) & \ni & \sigma\ (\mathit{OD} : (s : S) \to \mathsf{ROrnDesc}\ J\ e\ (D\ s)) \\ & \mid & \nabla\ (s : S)\ (\mathit{OD} : \mathsf{ROrnDesc}\ J\ e\ (D\ s)) \\ \mathsf{ROrnDesc}\ J\ e\ D & \ni & \Delta\ (T : \mathsf{Set})\ (\mathit{OD} : T \to \mathsf{ROrnDesc}\ J\ e\ D) \end{array} \end{array}$$

If the template response description is $\mathsf{v}\ \mathit{is}$, then we can specify a list of indices refining *is* (by v); if it is $\sigma\ S\ D$, then we can either copy (σ) or delete (∇) the field; finally, whatever the template is, we can always choose to create (Δ) a new field. This thesis maintains a separation between Agda datatypes and index-first datatypes, in particular using the former to construct a universe for the latter, but it is conceivable that ornaments and ornamental descriptions can be incorporated into a type theory with self-encoding index-first universes like the one presented by Chapman et al. [2010]. □

```
record _⋈_ {I J K : Set} (e : J → I) (f : K → I) : Set where
  constructor _,_
  field
    {i} : I   -- implicit field
    j   : e ⁻¹ i
    k   : f ⁻¹ i

pull : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → I
pull = _⋈_.i

outl⋈ : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → J
outl⋈ = und ∘ _⋈_.j

outr⋈ : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → K
outr⋈ = und ∘ _⋈_.k
```

Figure 3.2 Definitions for set-theoretic pullbacks.

3.2.3 Parallel composition of ornaments

Recall that our purpose of introducing ornaments is to be able to compute composite datatypes like ordered vectors. This can be achieved by composing two ornaments from the same description **in parallel**. The generic scenario is as follows (think of the direction of an ornamental arrow as following its forgetful function):

$$
\begin{tikzcd}
 & e \bowtie f \arrow[dl, "outl_{\bowtie}"'] \arrow[dd, "pull"] \arrow[dr, "outr_{\bowtie}"] & \\
J \arrow[dr, "e"'] & & K \arrow[dl, "f"] \\
 & I &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \lfloor O \otimes P \rfloor : \mathsf{Desc}\ e \bowtie f \arrow[dl, "diffOrn\text{-}l\ O\ P"'] \arrow[dd, "\lceil O \otimes P \rceil"] \arrow[dr, "diffOrn\text{-}r\ O\ P"] & \\
E : \mathsf{Desc}\ J \arrow[dr, "O"'] & & F : \mathsf{Desc}\ K \arrow[dl, "P"] \\
 & D : \mathsf{Desc}\ I &
\end{tikzcd}
$$

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

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

where $e \bowtie f$ is the set-theoretic pullback of $e : J \to I$ and $f : K \to I$, i.e., it is isomorphic to $\Sigma[\,jk : J \times K\,]\ e\ (outl\ jk) \equiv f\ (outr\ jk)$; related definitions are shown in Figure 3.2. 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 $\lceil O \otimes P \rceil$. 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**

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

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

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

and

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

are extensionally equal to $forget\ \lceil O \otimes P \rceil$. (The diagrams foreshadow our characterisation of parallel composition as a category-theoretic pullback in **??**; we will make their meanings precise there.)

Example (*ordered vectors*)**.** Consider the two ornaments $\lceil \mathit{OrdListOD} \rceil$ from lists to ordered lists and *ListD-VecD Val* from lists to vectors. Composing them in parallel gives us an ornamental description

$$\begin{array}{l} \mathit{OrdVecOD} : \mathsf{OrnDesc}\ (! \bowtie !)\ pull\ (\mathit{ListD}\ \mathit{Val}) \\ \mathit{OrdVecOD} = \lceil \mathit{OrdListOD} \rceil \otimes \mathit{ListD\text{-}VecD}\ \mathit{Val} \end{array}$$

from which we can decode a new datatype of ordered vectors by

$$\begin{array}{l} \mathsf{OrdVec} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set} \\ \mathsf{OrdVec}\ b\ n = \mu\ \lfloor \mathit{OrdVecOD} \rfloor\ (\mathsf{ok}\ (\mathsf{ok}\ b\ , \mathsf{ok}\ n)) \end{array}$$

and an ornament $\lceil \mathit{OrdVecOD} \rceil$ 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

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

- 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.

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

  (The implicit argument $\{s := s\}$ is the one named s in the type of *pcROD-double*∇ — the 's' to the left of ':=' is the name of the argument, while the 's' to the right is a

```
fromEq : {I J : Set} (e : J → I) {j : J} {i : I} → e j ≡ i → e ⁻¹ i
fromEq e {j} refl = ok j

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.

pattern variable. This syntax allows us to skip the nine implicit arguments before this one.) 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

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

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* $\rceil$ and *ListD-VecD Val* in parallel yields the following ornamental description relative to *ListD Val*:

$$\begin{array}{lll}
\lambda\ \{\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok\ zero}\quad)) & \mapsto & \boxed{\nabla[\,\text{`}\mathsf{nil}\,]}\ \mathsf{v}\ \blacksquare \\
\ \ ;\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\mathsf{suc}\ n))) & \mapsto & \boxed{\nabla[\,\text{`}\mathsf{cons}\,]}\ \sigma[\,x : \mathit{Val}\,] \\
 & & \quad \boxed{\Delta[\,\mathit{leq} : b \leqslant x\,]}\ \mathsf{v}\ (\mathsf{ok}\ (\boxed{\mathsf{ok}\ x}\ ,\ \boxed{\mathsf{ok}\ n})\ ,\ \blacksquare)\ \}
\end{array}$$

where lighter box indicates modifications from $\lceil$ *OrdListOD* $\rceil$ and darker box from *ListD-VecD Val*. □

Finally, the definitions for left difference ornament are shown in Figure 3.4. Left difference ornament has the same structure as parallel composition, but

$$
\begin{array}{l}
\mathit{und\text{-}fromEq} \;:\\
\quad \{I\,J \,:\, \mathsf{Set}\}\ (e \,:\, J \to I)\ \{j \,:\, J\}\ \{I \,:\, I\}\ (\mathit{eq} \,:\, e\ j \equiv i) \to \mathit{und}\ (\mathit{fromEq}\ e\ \mathit{eq}) \equiv j\\
\mathit{und\text{-}fromEq}\ e\ \mathsf{refl} \;=\; \mathsf{refl}
\end{array}
$$

$$
\begin{array}{l}
\mathit{diff\text{-}}\mathbb{E}\mathit{\text{-}l} \,:\, \{I\,J\,K \,:\, \mathsf{Set}\}\ \{e \,:\, J \to I\}\ \{f \,:\, K \to I\} \to\\
\qquad\qquad \{\mathit{is} \,:\, \mathsf{List}\ I\}\ \{\mathit{js} \,:\, \mathsf{List}\ J\}\ \{\mathit{ks} \,:\, \mathsf{List}\ K\} \to\\
\qquad\qquad (\mathit{eeqs} \,:\, \mathbb{E}\ e\ \mathit{js}\ \mathit{is})\ (\mathit{feqs} \,:\, \mathbb{E}\ f\ \mathit{ks}\ \mathit{is}) \to \mathbb{E}\ \mathit{outl}_{\bowtie}\ (\mathit{und\text{-}}\mathbb{P}\ \mathit{is}\ (\mathit{pc\text{-}}\mathbb{E}\ \mathit{eeqs}\ \mathit{feqs}))\ \mathit{js}\\
\mathit{diff\text{-}}\mathbb{E}\mathit{\text{-}l}\qquad\quad []\qquad\qquad\quad []\qquad\qquad\ = []\\
\mathit{diff\text{-}}\mathbb{E}\mathit{\text{-}l}\ \{e := e\}\ (\mathit{eeq} :: \mathit{eeqs})\ (\mathit{feq} :: \mathit{feqs}) \;=\; \mathit{und\text{-}fromEq}\ e\ \mathit{eeq} :: \mathit{diff\text{-}}\mathbb{E}\mathit{\text{-}l}\ \mathit{eeqs}\ \mathit{feqs}
\end{array}
$$

mutual

$$
\begin{array}{l}
\quad \mathit{diffROrn\text{-}l} \;:\\
\qquad \{I\,J\,K \,:\, \mathsf{Set}\}\ \{e \,:\, J \to I\}\ \{f \,:\, K \to I\} \to\\
\qquad \{D \,:\, \mathsf{RDesc}\ I\}\ \{E \,:\, \mathsf{RDesc}\ J\}\ \{F \,:\, \mathsf{RDesc}\ K\} \to\\
\qquad (O \,:\, \mathsf{ROrn}\ e\ D\ E)\ (P \,:\, \mathsf{ROrn}\ f\ D\ F) \to \mathsf{ROrn}\ \mathit{outl}_{\bowtie}\ E\ (\mathit{toRDesc}\ (\mathit{pcROD}\ O\ P))\\
\quad \mathit{diffROrn\text{-}l}\ (\mathsf{v}\ \mathit{eeqs})\ \ (\mathsf{v}\ \mathit{feqs}) \;=\; \mathsf{v}\ (\mathit{diff\text{-}}\mathbb{E}\mathit{\text{-}l}\ \mathit{eeqs}\ \mathit{feqs})\\
\quad \mathit{diffROrn\text{-}l}\ (\mathsf{v}\ \mathit{eeqs})\ \ (\Delta\ T\ P) \;=\; \Delta[\,t : T\,]\ \mathit{diffROrn\text{-}l}\ (\mathsf{v}\ \mathit{eeqs})\ \ (P\ t)\\
\quad \mathit{diffROrn\text{-}l}\ (\sigma\ S\ O)\ \ (\sigma\ .S\ P) \;=\; \sigma[\,s : S\,]\ \mathit{diffROrn\text{-}l}\ (O\ s)\quad (P\ s)\\
\quad \mathit{diffROrn\text{-}l}\ (\sigma\ S\ O)\ \ (\Delta\ T\ P) \;=\; \Delta[\,t : T\,]\ \mathit{diffROrn\text{-}l}\ (\sigma\ S\ O)\ (P\ t)\\
\quad \mathit{diffROrn\text{-}l}\ (\sigma\ S\ O)\ \ (\nabla\ s\ P) \;=\; \nabla[\,s\,]\quad\ \mathit{diffROrn\text{-}l}\ (O\ s)\quad P\\
\quad \mathit{diffROrn\text{-}l}\ (\Delta\ T\ O)\ P \qquad\ \;=\; \sigma[\,t : T\,]\ \mathit{diffROrn\text{-}l}\ (O\ t)\quad P\\
\quad \mathit{diffROrn\text{-}l}\ (\nabla\ s\ O)\ \ (\sigma\ S\ P) \;=\; \qquad\qquad \mathit{diffROrn\text{-}l}\ O\qquad (P\ s)\\
\quad \mathit{diffROrn\text{-}l}\ (\nabla\ s\ O)\ \ (\Delta\ T\ P) \;=\; \Delta[\,t : T\,]\ \mathit{diffROrn\text{-}l}\ (\nabla\ s\ O)\ (P\ t)\\
\quad \mathit{diffROrn\text{-}l}\ (\nabla\ s\ O)\ \ (\nabla\ s'\ P) \;=\; \Delta\ (s \equiv s')\ (\mathit{diffROrn\text{-}l\text{-}double}\nabla\ O\ P)
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{diffROrn\text{-}l\text{-}double}\nabla \;:\\
\qquad \{I\,J\,K\,S \,:\, \mathsf{Set}\}\ \{e \,:\, J \to I\}\ \{f \,:\, K \to I\} \to\\
\qquad \{D \,:\, S \to \mathsf{RDesc}\ I\}\ \{E \,:\, \mathsf{RDesc}\ J\}\ \{F \,:\, \mathsf{RDesc}\ K\}\ \{s\ s' \,:\, S\} \to\\
\qquad (O \,:\, \mathsf{ROrn}\ e\ (D\ s)\ E)\ (P \,:\, \mathsf{ROrn}\ f\ (D\ s')\ F)\ (\mathit{eq} \,:\, s \equiv s') \to\\
\qquad \mathsf{ROrn}\ \mathit{outl}_{\bowtie}\ E\ (\mathit{toRDesc}\ (\mathit{pcROD\text{-}double}\nabla\ O\ P\ \mathit{eq}))\\
\quad \mathit{diffROrn\text{-}l\text{-}double}\nabla\ O\ P\ \mathsf{refl} \;=\; \mathit{diffROrn\text{-}l}\ O\ P
\end{array}
$$

$$
\begin{array}{l}
\mathit{diffOrn\text{-}l} \,:\, \{I\,J\,K \,:\, \mathsf{Set}\}\ \{e \,:\, J \to I\}\ \{f \,:\, K \to I\} \to\\
\qquad\qquad \{D \,:\, \mathsf{Desc}\ I\}\ \{E \,:\, \mathsf{Desc}\ J\}\ \{F \,:\, \mathsf{Desc}\ K\} \to\\
\qquad\qquad (O \,:\, \mathsf{Orn}\ e\ D\ E)\ (P \,:\, \mathsf{Orn}\ f\ D\ F) \to \mathsf{Orn}\ \mathit{outl}_{\bowtie}\ E\ \lfloor O \otimes P \rfloor\\
\mathit{diffOrn\text{-}l}\ O\ P\ (\mathsf{ok}\ (j\,,k)) \;=\; \mathit{diffROrn\text{-}l}\ (O\ j)\ (P\ k)
\end{array}
$$

Figure 3.4 Definitions for left difference ornament.

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 *forget O* removes modifications encoded in the left-hand side ornament. Right difference ornament is defined analogously and is omitted from the presentation.

3.3 Refinement semantics of ornaments

We now know how to do function upgrading with refinements (Section 3.1) and how to relate datatypes and manufacture composite datatypes with ornaments (Section 3.2), and there is only one link missing: translation of ornaments to refinements. Every ornament $O\ :\ \mathsf{Orn}\ e\ D\ E$ induces a refinement family from $\mu\ D$ to $\mu\ E$. That is, we can construct

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

which is called the **refinement semantics** of ornaments (Section 3.3.1). The construction of *RSem* begins in Section 3.3.1 and continues into **??** (where we introduce a lightweight categorical organisation). Another important aspect of the translation is from parallel composition of ornaments to refinements whose promotion predicate is pointwise conjunctive. This begins in Section 3.3.2 and also continues into **??**.

3.3.1 Optimised predicates

Our task in this section 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 $x \;:\; \mu\;D\;i$, a proof of type $\mathsf{OptP}\;\;O\;\;j\;\;x$ contains information for complementing x and forming an inhabitant y of type $\mu\;E\;(\mathit{und}\;j)$ 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

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

to a list, an optimised promotion proof would look like

$$\begin{aligned}p \;=\; & \mathsf{con}\;(a\;\;, \\ & \mathsf{con}\;(a'\;, \\ & \mathsf{con}\;(\\ & \quad\blacksquare)\;,\;\blacksquare)\;,\;\blacksquare)\;\;:\;\;\mathsf{OptP}\;(\mathit{NatD\text{-}ListD}\;A)\;(\mathsf{ok}\;\blacksquare)\;\mathit{two}\end{aligned}$$

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

$$\begin{aligned}& \mathsf{con}\;(\text{`}\mathsf{cons}\;,\;a\;\;, \\ & \mathsf{con}\;(\text{`}\mathsf{cons}\;,\;a'\;,\end{aligned}$$

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

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 manner, 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 $\lceil \mathit{singletonOD}\ D \rceil$:

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

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

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

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*:

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

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

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

$$
\begin{array}{ll}
\lambda\ \{\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ []\ )))\ \mapsto & \boxed{\nabla[\,\text{`nil}\,]}\ \mathsf{v}\ \blacksquare \\
\ \ ;\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ (\blacksquare\ ,\ x :: xs)))\ \mapsto & \boxed{\nabla[\,\text{`cons}\,]}\ \boxed{\nabla[\,x\,]}\ \boxed{\Delta[\,leq : b \leqslant x\,]} \\
 & \quad \mathsf{v}\ (\mathsf{ok}\ (\boxed{\mathsf{ok}\ x}\ ,\ \boxed{\mathsf{ok}\ (\blacksquare\ ,\ xs)}\ )\ ,\ \blacksquare)\ \}
\end{array}
$$

where lighter box indicates contributions from $\lceil \mathit{OrdListOD} \rceil$ and darker box from $\lceil \mathit{singletonOD}\ (\mathit{ListD}\ \mathit{Val}) \rceil$. Since a proof of $\mathsf{Ordered}\ b\ xs$ consists of exactly the inequality proofs necessary for ensuring that xs 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 $\mathit{ListD\text{-}VecD}\ A$ from lists to vectors is produced by the ornamental description $\mathit{OptPOD}\ (\mathit{ListD\text{-}VecD}\ A)$ relative to $\mathit{ListD}\ A$:

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

where lighter box indicates contributions from $\mathit{ListD\text{-}VecD}\ A$ and darker box from $\lceil\ \mathit{singletonOD}\ (\mathit{ListD}\ A)\ \rceil$. 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:

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

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

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

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

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

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

The construction of *ornConvIso* is deferred to **??**.

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 $\lceil \lceil \mathit{OrdListOD} \rceil \otimes \mathit{ListD\text{-}VecD}\ \mathit{Val} \rceil$ from lists to ordered vectors is

$$\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{indexfirst data}\ \mathsf{OrderedLength} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{List}\ \mathit{Val} \to \mathsf{Set}\ \textbf{where}} \\
\quad \mathsf{OrderedLength}\ b\ \mathsf{zero} & [] & \ni & \mathsf{nil} \\
\quad \mathsf{OrderedLength}\ b\ \mathsf{zero} & (x :: xs) & \not\ni & \\
\quad \mathsf{OrderedLength}\ b\ (\mathsf{suc}\ n) & [] & \not\ni & \\
\quad \mathsf{OrderedLength}\ b\ (\mathsf{suc}\ n) & (x :: xs) & \ni & \mathsf{cons}\ (\mathit{leq} : b \leqslant x) \\
 & & & \qquad\qquad (\mathit{ol} : \mathsf{OrderedLength}\ x\ n\ xs)
\end{array}$$

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

$$\mathsf{OrderedLength}\ b\ n\ xs \cong \mathsf{Ordered}\ b\ xs \times \mathsf{Length}\ n\ xs$$

for all $b : \mathit{Val}$, $n : \mathsf{Nat}$, and $xs : \mathsf{List}\ \mathit{Val}$ — 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 is deferred to **??**. 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:

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Swap}\ \{A\ B\ :\ \mathsf{Set}\}\ (r\ :\ \mathsf{Refinement}\ A\ B)\ :\ \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad P\ :\ A \rightarrow \mathsf{Set} \\
\qquad i\ \ :\ (a\ :\ A) \rightarrow \mathsf{Refinement}.P\ r\ a\ \cong\ P\ a
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{toRefinement}\ :\ \{A\ B\ :\ \mathsf{Set}\}\ \{r\ :\ \mathsf{Refinement}\ A\ B\} \rightarrow \mathsf{Swap}\ r \rightarrow \mathsf{Refinement}\ A\ B \\
\mathit{toRefinement}\ s\ =\ \textbf{record}\ \{\ P\ =\ \mathsf{Swap}.P\ s \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ i\ \ =\ \{\ \}_0\ \}
\end{array}
$$

where Goal 0 is the new conversion isomorphism

$$
B\ \cong\ \Sigma\ A\ (\mathsf{Refinement}.P\ r)\ \cong\ \Sigma\ A\ (\mathsf{Swap}.P\ s)
$$

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:

$$
\begin{array}{l}
\mathsf{FSwap}\ :\ \{I\ J\ :\ \mathsf{Set}\}\ \{e\ :\ J \rightarrow I\}\ \{X\ :\ I \rightarrow \mathsf{Set}\}\ \{Y\ :\ J \rightarrow \mathsf{Set}\} \rightarrow \\
\qquad\qquad (rs\ :\ \mathsf{FRefinement}\ e\ X\ Y) \rightarrow \mathsf{Set}_1 \\
\mathsf{FSwap}\ rs\ =\ \{i\ :\ I\}\ (j\ :\ e^{\ -1}\ i) \rightarrow \mathsf{Swap}\ (rs\ j)
\end{array}
$$

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 $\lceil O \otimes P \rceil$:

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

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

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

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

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

where

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

simply retains the original promotion predicate in rs. □

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

$$
\begin{aligned}
&\mathit{OrdVec\text{-}FSwap} : \mathsf{FSwap}\ (\mathit{RSem}\ \lceil \mathit{OrdVecOD} \rceil) \\
&\mathit{OrdVec\text{-}FSwap} = \\
&\quad \otimes\text{-}\mathit{FSwap}\ \lceil \mathit{OrdListOD} \rceil\ (\mathit{ListD\text{-}VecD}\ \mathit{Val})\ \mathit{id\text{-}FSwap}\ (\mathit{Length\text{-}FSwap}\ \mathit{Val})
\end{aligned}
$$

we can extract a refinement family from lists to ordered vectors using

$$\lambda\, b\ n\ \mathit{xs} \mapsto \mathsf{Ordered}\ b\ \mathit{xs} \times \mathit{length}\ \mathit{xs} \equiv n$$

as the promotion predicate, where

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

swaps Length for $\lambda\, n\ xs \mapsto \mathit{length}\ xs \ \equiv\ n$. $\square$

3.4 Examples

To demonstrate the use of the ornament–refinement framework, in Section 3.4.1 we first conclude the example about insertion into a list introduced in **??**, and then we 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** (Section 3.4.2) 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** (Section 3.4.3) demonstrates the power of parallel composition of ornaments by treating heap ordering and leftist balancing properties modularly.

3.4.1 Insertion into a list

To recap: we have an externalist library for lists which supports one operation

$$\mathit{insert}\ :\ \mathit{Val} \to \mathsf{List}\ \mathit{Val} \to \mathsf{List}\ \mathit{Val}$$

and has two modules about length and ordering, respectively containing the following two proofs about *insert*:

$$\begin{aligned}
\mathit{insert\text{-}length}\ \ &:\ (y\ :\ \mathit{Val})\ \{n\ :\ \mathsf{Nat}\}\ (xs\ :\ \mathsf{List}\ \mathit{Val}) \to \\
&\quad\ \mathit{length}\ xs\ \equiv\ n \to \mathit{length}\ (\mathit{insert}\ y\ xs)\ \equiv\ \mathsf{suc}\ n \\
\mathit{insert\text{-}ordered}\ &:\ (y\ :\ \mathit{Val})\ \{b\ :\ \mathit{Val}\}\ (xs\ :\ \mathsf{List}\ \mathit{Val}) \to \mathsf{Ordered}\ b\ xs \to \\
&\quad\ \{b'\ :\ \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \mathsf{Ordered}\ b'\ (\mathit{insert}\ y\ xs)
\end{aligned}$$

-- the upgraded function type has an extra argument

$$
\begin{aligned}
&\mathit{new} : \{A : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\} \to \\
&\qquad ((i : I) \to \mathsf{Upgrade}\ A\ (X\ i)) \to \mathsf{Upgrade}\ A\ ((i : I) \to X\ i) \\
&\mathit{new}\ I\ u = \mathbf{record}\ \{\ P = \lambda\, a \mapsto (i : I) \to \mathsf{Upgrade}.P\ (u\ i)\ a \\
&\qquad\qquad\qquad\ ;\ C = \lambda\, a\ x \mapsto (i : I) \to \mathsf{Upgrade}.C\ (u\ i)\ a\ (x\ i) \\
&\qquad\qquad\qquad\ ;\ u = \lambda\, a\ p\ i \mapsto \mathsf{Upgrade}.u\ (u\ i)\ a\ (p\ i) \\
&\qquad\qquad\qquad\ ;\ c = \lambda\, a\ p\ i \mapsto \mathsf{Upgrade}.c\ (u\ i)\ a\ (p\ i)\ \}
\end{aligned}
$$

$$\mathbf{syntax}\ \mathit{new}\ I\ (\lambda\, i \mapsto u) = \forall^{+}[\, i : I\,]\ u$$

-- implicit version of *new*

$$
\begin{aligned}
&\mathit{new}' : \{A : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\} \to \\
&\qquad ((i : I) \to \mathsf{Upgrade}\ A\ (X\ i)) \to \mathsf{Upgrade}\ A\ (\{i : I\} \to X\ i) \\
&\mathit{new}'\ I\ u = \mathbf{record}\ \{\ P = \lambda\, a \mapsto \{i : I\} \to \mathsf{Upgrade}.P\ (u\ i)\ a \\
&\qquad\qquad\qquad\ ;\ C = \lambda\, a\ x \mapsto \{i : I\} \to \mathsf{Upgrade}.C\ (u\ i)\ a\ (x\ \{i\}) \\
&\qquad\qquad\qquad\ ;\ u = \lambda\, a\ p\ \{i\} \mapsto \mathsf{Upgrade}.u\ (u\ i)\ a\ (p\ \{i\}) \\
&\qquad\qquad\qquad\ ;\ c = \lambda\, a\ p\ \{i\} \mapsto \mathsf{Upgrade}.c\ (u\ i)\ a\ (p\ \{i\})\ \}
\end{aligned}
$$

$$\mathbf{syntax}\ \mathit{new}'\ I\ (\lambda\, i \mapsto u) = \forall^{+}[[\, i : I\,]]\ u$$

-- the underlying and the upgraded function types have a common argument

$$
\begin{aligned}
&\mathit{fixed} : (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\}\ \{Y : I \to \mathsf{Set}\} \to \\
&\qquad ((i : I) \to \mathsf{Upgrade}\ (X\ i)\ (Y\ i)) \to \mathsf{Upgrade}\ ((i : I) \to X\ i)\ ((i : I) \to Y\ i) \\
&\mathit{fixed}\ I\ u = \mathbf{record}\ \{\ P = \lambda f \mapsto (i : I) \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ i) \\
&\qquad\qquad\qquad\ ;\ C = \lambda f\ g \mapsto (i : I) \to \mathsf{Upgrade}.C\ (u\ i)\ (f\ i)\ (g\ i) \\
&\qquad\qquad\qquad\ ;\ u = \lambda f\ h\ i \mapsto \mathsf{Upgrade}.u\ (u\ i)\ (f\ i)\ (h\ i) \\
&\qquad\qquad\qquad\ ;\ c = \lambda f\ h\ i \mapsto \mathsf{Upgrade}.c\ (u\ i)\ (f\ i)\ (h\ i)\ \}
\end{aligned}
$$

$$\mathbf{syntax}\ \mathit{fixed}\ I\ (\lambda\, i \mapsto u) = \forall[\, i : I\,]\ u$$

-- implicit version of *fixed*

$$
\begin{aligned}
&\mathit{fixed}' : (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\}\ \{Y : I \to \mathsf{Set}\} \to \\
&\qquad ((i : I) \to \mathsf{Upgrade}\ (X\ i)\ (Y\ i)) \to \mathsf{Upgrade}\ (\{i : I\} \to X\ i)\ (\{i : I\} \to Y\ i) \\
&\mathit{fixed}'\ I\ u = \mathbf{record}\ \{\ P = \lambda f \mapsto \{i : I\} \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ \{i\}) \\
&\qquad\qquad\qquad\ ;\ C = \lambda f\ g \mapsto \{i : I\} \to \mathsf{Upgrade}.C\ (u\ i)\ (f\ \{i\})\ (g\ \{i\}) \\
&\qquad\qquad\qquad\ ;\ u = \lambda f\ h\ \{i\} \mapsto \mathsf{Upgrade}.u\ (u\ i)\ (f\ \{i\})\ (h\ \{i\}) \\
&\qquad\qquad\qquad\ ;\ c = \lambda f\ h\ \{i\} \mapsto \mathsf{Upgrade}.c\ (u\ i)\ (f\ \{i\})\ (h\ \{i\})\ \}
\end{aligned}
$$

$$\mathbf{syntax}\ \mathit{fixed}'\ I\ (\lambda\, i \mapsto u) = \forall[[\, i : I\,]]\ u$$

Figure 3.5 More combinators on upgrades.

To upgrade the library to also work as an internalist one, all we have to do is add to the two modules the descriptions of vectors and ordered lists and the ornaments from lists to vectors and ordered lists (or equivalently and more simply, just the ornamental descriptions). Now we can manufacture

$$\mathit{insert}_V : \mathit{Val} \to \{n : \mathsf{Nat}\} \to \mathsf{Vec}\ \mathit{Val}\ n \to \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n)$$

starting with writing the following upgrade, which marks how the types of *insert* and insert_V are related:

$$\begin{aligned}
&\mathit{upg} : \mathsf{Upgrade}\ (\mathit{Val} \to \phantom{\{n : \mathsf{Nat}\} \to}\ \mathsf{List}\ \mathit{Val} \to \mathsf{List}\ \mathit{Val} \phantom{(\mathsf{suc}\ n)}\ )\\
&\qquad\qquad\qquad (\mathit{Val} \to \{n : \mathsf{Nat}\} \to \mathsf{Vec}\ \mathit{Val}\ n \to \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n))\\
&\mathit{upg} = \forall[\,_ : \mathit{Val}\,]\ \forall^{+}[[\,n : \mathsf{Nat}\,]]\ r\ n \rightharpoonup \mathit{toUpgrade}\ (r\ (\mathsf{suc}\ n))\\
&\quad \mathbf{where}\ r : (n : \mathsf{Nat}) \to \mathsf{Refinement}\ (\mathsf{List}\ \mathit{Val})\ (\mathsf{Vec}\ \mathit{Val}\ n)\\
&\qquad\qquad r\ n = \mathit{toRefinement}\ (\mathit{Length\text{-}FSwap}\ \mathit{Val}\ (\mathsf{ok}\ n))
\end{aligned}$$

That is, the type of insert_V has a common first argument with the type of *insert* and a new implicit argument $n : \mathsf{Nat}$, and refines the two occurrences of $\mathsf{List}\ \mathit{Val}$ in the type of *insert* to $\mathsf{Vec}\ \mathit{Val}\ n$ and $\mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n)$. The function insert_V is then simply defined by

$$\begin{aligned}
&\mathit{insert}_V : \mathit{Val} \to \{n : \mathsf{Nat}\} \to \mathsf{Vec}\ \mathit{Val}\ n \to \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n)\\
&\mathit{insert}_V = \mathsf{Upgrade}.u\ \mathit{upg}\ \mathit{insert}\ \mathit{insert\text{-}length}
\end{aligned}$$

which satisfies the coherence property

$$\begin{aligned}
&\mathit{insert}_V\text{-}\mathit{coherence} :\\
&\quad (y : \mathit{Val})\ \{n : \mathsf{Nat}\}\ (\mathit{xs} : \mathsf{List}\ \mathit{Val})\ (\mathit{xs}' : \mathsf{Vec}\ \mathit{Val}\ n) \to\\
&\quad \mathit{forget}\ (\mathit{ListD\text{-}VecD}\ \mathit{Val})\ \mathit{xs}' \equiv \mathit{xs} \to\\
&\quad \mathit{forget}\ (\mathit{ListD\text{-}VecD}\ \mathit{Val})\ (\mathit{insert}_V\ y\ \mathit{xs}') \equiv \mathit{insert}\ y\ \mathit{xs}\\
&\mathit{insert}_V\text{-}\mathit{coherence} = \mathsf{Upgrade}.c\ \mathit{upg}\ \mathit{insert}\ \mathit{insert\text{-}length}
\end{aligned}$$

That is, insert_V manipulates the underlying list of the input vector in the same way as *insert*. Similarly we can manufacture insert_O for ordered lists by using an appropriate upgrade that accepts *insert-ordered* as a promotion proof for *insert*. For ordered vectors, the datatype is manufactured by parallel composition, and the operation

$$\begin{aligned}&\mathit{insert}_{OV} \;:\; (y \;:\; \mathit{Val})\; \{b \;:\; \mathit{Val}\}\; \{n \;:\; \mathsf{Nat}\} \to \mathsf{OrdVec}\; b\; n \to \\ &\qquad\qquad \{b' \;:\; \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \mathsf{OrdVec}\; b'\; (\mathsf{suc}\; n)\end{aligned}$$

is manufactured with the help of the upgrade

$$\begin{aligned}&\forall[\, y : \mathit{Val} \,]\;\; \forall^{+}[[\, b : \mathit{Val} \,]]\;\; \forall^{+}[[\, n : \mathsf{Nat} \,]]\;\; r\; b\; n \rightharpoonup \\ &\forall^{+}[[\, b' : \mathit{Val} \,]]\;\; \forall^{+}[\, \_ : b' \leqslant y \,]\;\; \forall^{+}[\, \_ : b' \leqslant b \,]\;\; \mathit{toUpgrade}\; (r\; b'\; (\mathsf{suc}\; n)) \\ &\textbf{where} \\ &\quad r \;:\; (b \;:\; \mathit{Val})\; (n \;:\; \mathsf{Nat}) \to \mathsf{Refinement}\; (\mathsf{List}\; \mathit{Val})\; (\mathsf{OrdVec}\; b\; n) \\ &\quad r\; b\; n \;=\; \mathit{toRefinement}\; (\mathit{OrdVec\text{-}FSwap}\; (\mathsf{ok}\; (\mathsf{ok}\; b \,,\, \mathsf{ok}\; n)))\end{aligned}$$

The type of promotion proofs for *insert* specified by this upgrade is

$$\begin{aligned}&(y \;:\; \mathit{Val})\; \{b \;:\; \mathit{Val}\}\; \{n \;:\; \mathsf{Nat}\}\; (\mathit{xs} \;:\; \mathsf{List}\; \mathit{Val}) \to \\ &\quad \mathsf{Ordered}\; b\; \mathit{xs} \times \mathit{length}\; \mathit{xs} \;\equiv\; n \to \\ &\{b' \;:\; \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \\ &\quad \mathsf{Ordered}\; b'\; (\mathit{insert}\; y\; \mathit{xs}) \times \mathit{length}\; (\mathit{insert}\; y\; \mathit{xs}) \;\equiv\; \mathsf{suc}\; n\end{aligned}$$

and is inhabited by

$$\begin{aligned}\lambda\; \{\; y\; \mathit{xs}\; (\mathit{ord} \,,\, \mathit{len})\; b'{\leqslant}y\; b'{\leqslant}b \;\mapsto\;& \mathit{insert\text{-}ordered}\; y\; \mathit{xs}\; \mathit{ord}\; b'{\leqslant}y\; b'{\leqslant}b \,, \\ & \mathit{insert\text{-}length}\;\; y\; \mathit{xs}\; \mathit{len} \;\}\end{aligned}$$

which is strikingly similar to insert_{EOV} in **??**.

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 3.6) 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

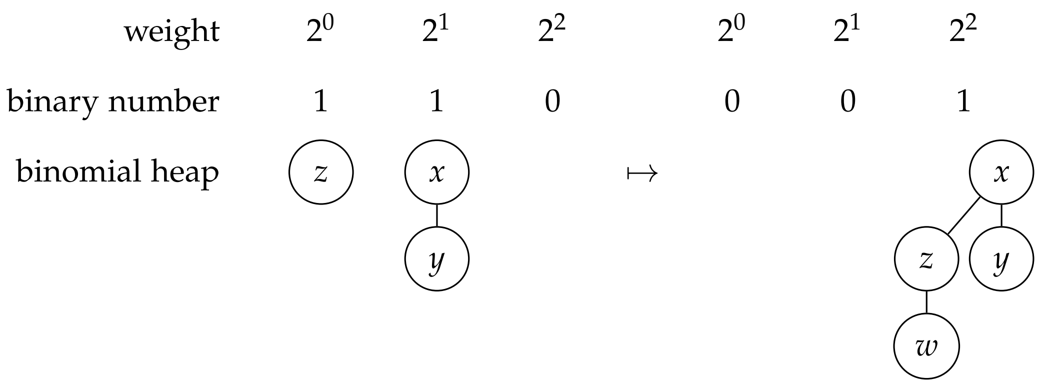

Figure 3.6 **Left:** a binomial heap of size 3 consisting of two binomial trees storing elements x, y, and z. **Right:** a possible result of inserting an element w into the heap. (Note that the digits of the underlying binary numbers are ordered with the least significant digit first.)

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**. 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.

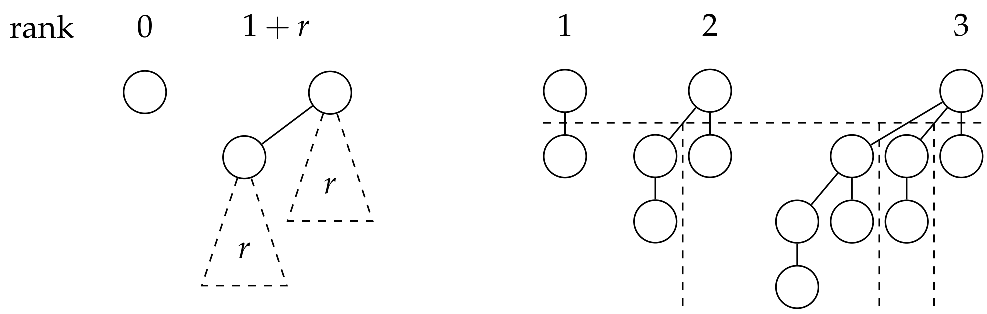

Figure 3.7 **Left:** inductive definition of binomial trees. **Right:** decomposition of binomial trees of ranks 1 to 3.

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 3.7):

- 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 3.7 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:

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

$$\mathsf{BTree} \;=\; \mu\; \mathit{BTreeD}$$

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

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

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.

$$\begin{aligned}
&\mathit{attach} : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\;r \to \mathsf{BTree}\;r \to \mathsf{BTree}\;(\mathsf{suc}\;r) \\
&\mathit{attach}\;t\;(\mathsf{con}\;(y\;,\;us)) \;=\; \mathsf{con}\;(y\;,\;t\;,\;us)
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{link} : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\;r \to \mathsf{BTree}\;r \to \mathsf{BTree}\;(\mathsf{suc}\;r) \\
&\mathit{link}\;t\;u\;\mathbf{with}\;\mathit{root}\;t \leqslant_?\; \mathit{root}\;u \\
&\mathit{link}\;t\;u \mid \mathsf{yes}\;_ \;=\; \mathit{attach}\;u\;t \\
&\mathit{link}\;t\;u \mid \mathsf{no}\;\;_ \;=\; \mathit{attach}\;t\;u
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{root} : \{r : \mathsf{Nat}\} \to \mathsf{BTree}\;r \to \mathit{Val} \\
&\mathit{root}\;(\mathsf{con}\;(x\;,\;ts)) \;=\; x
\end{aligned}$$

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 $\mathsf{Bin} : \mathsf{Set}$ of binary numbers is just a specialised datatype of lists of binary digits:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{BinTag} : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{`nil} \ \ : \mathsf{BinTag} \\
\quad \mathsf{`zero} : \mathsf{BinTag} \\
\quad \mathsf{`one} \ : \mathsf{BinTag}
\end{array}
$$

$$
\begin{array}{l}
\mathit{BinD} : \mathsf{Desc}\ \top \\
\mathit{BinD}\ \blacksquare = \sigma\ \mathsf{BinTag}\ \lambda\ \{\ \mathsf{`nil} \mapsto \mathsf{v}\ [\,] \\
\qquad\qquad\qquad\qquad\qquad ;\ \mathsf{`zero} \mapsto \mathsf{v}\ (\blacksquare :: [\,]) \\
\qquad\qquad\qquad\qquad\qquad ;\ \mathsf{`one} \mapsto \mathsf{v}\ (\blacksquare :: [\,])\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Bin} : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Bin} \ni \mathsf{nil} \\
\qquad\quad \mid \mathsf{zero}\ (b : \mathsf{Bin}) \\
\qquad\quad \mid \mathsf{one}\ \ (b : \mathsf{Bin})
\end{array}
$$

The intended interpretation of binary numbers is given by

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{toNat} : \mathsf{Bin} \to \mathsf{Nat}} \\
\mathit{toNat}\ \mathsf{nil} & = & 0 \\
\mathit{toNat}\ (\mathsf{zero}\ b) & = & 0 + 2 * \mathit{toNat}\ b \\
\mathit{toNat}\ (\mathsf{one}\ \ b) & = & 1 + 2 * \mathit{toNat}\ b
\end{array}
$$

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 $\mathsf{BHeap} : \mathsf{Nat} \to \mathsf{Set}$ is indexed with a "starting rank": if a binomial heap of type $\mathsf{BHeap}\ r$ 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 $1 + r$.

$$
\begin{aligned}
&\mathit{BHeapOD} : \mathsf{OrnDesc}\ \mathsf{Nat}\ !\ \mathit{BinD}\\
&\mathit{BHeapOD}\ (\mathsf{ok}\ r) = \sigma\ \mathsf{BinTag}\ \lambda\ \{\ \text{'nil} \mapsto \mathsf{v}\ \blacksquare\\
&\qquad\qquad\qquad\qquad\qquad\qquad ;\ \text{'zero} \mapsto \mathsf{v}\ (\mathsf{ok}\ (\mathsf{suc}\ r)\ ,\ \blacksquare)\\
&\qquad\qquad\qquad\qquad\qquad\qquad ;\ \text{'one} \mapsto \Delta[\,t : \mathsf{BTree}\ r\,]\ \mathsf{v}\ (\mathsf{ok}\ (\mathsf{suc}\ r)\ ,\ \blacksquare)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{BHeap} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{BHeap}\ r \ni \mathsf{nil}\\
&\qquad\qquad\quad \mid \mathsf{zero}\ (h : \mathsf{BHeap}\ (\mathsf{suc}\ r))\\
&\qquad\qquad\quad \mid \mathsf{one}\ \ (t : \mathsf{BTree}\ r)\ (h : \mathsf{BHeap}\ (\mathsf{suc}\ r))
\end{aligned}
$$

In applications, we would use binomial heaps of type $\mathsf{BHeap}\ 0$, which encompasses binomial heaps of all sizes.

Increment and insertion, in coherence

Increment of binary numbers is defined by

$$
\begin{aligned}
&\mathit{incr} : \mathsf{Bin} \to \mathsf{Bin}\\
&\mathit{incr}\ \mathsf{nil} &&= \mathsf{one}\ \mathsf{nil}\\
&\mathit{incr}\ (\mathsf{zero}\ b) &&= \mathsf{one}\ b\\
&\mathit{incr}\ (\mathsf{one}\ \ b) &&= \mathsf{zero}\ (\mathit{incr}\ b)
\end{aligned}
$$

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

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

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 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:

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

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 $\lceil \mathit{BHeapOD} \rceil$ (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

$$
\begin{array}{l}
\{r \,:\, \mathsf{Nat}\}\ (t \,:\, \mathsf{BTree}\ r)\ (b \,:\, \mathsf{Bin})\ (h \,:\, \mathsf{BHeap}\ r) \to \\
\mathit{toBin}\ h \;\equiv\; b \to \mathit{toBin}\ (\mathit{insT}\ t\ h) \;\equiv\; \mathit{incr}\ b
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{toBin} \;:\; \{r \,:\, \mathsf{Nat}\} \to \mathsf{BHeap}\ r \to \mathsf{Bin} \\
\mathit{toBin} \;=\; \mathit{forget}\ \lceil \mathit{BHeapOD} \rceil
\end{array}
$$

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

We have seen that the coherence property 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 $\mathsf{BHeap}\ r$:

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

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

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

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

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

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

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:

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

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*:

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

which is automatically in coherence with *incr*:

$$
\begin{array}{lcl}
\mathit{incr\text{-}insT\text{-}coherence} & : & \{r : \mathsf{Nat}\}\ (t : \mathsf{BTree}\ r)\ (b : \mathsf{Bin})\ (h : \mathsf{BHeap}\ r) \to \\
 & & \mathit{toBin}\ h \equiv b \to \mathit{toBin}\ (\mathit{insT}\ t\ h) \equiv \mathit{incr}\ b \\
\mathit{incr\text{-}insT\text{-}coherence} & = & \mathsf{Upgrade}.c\ \mathit{upg}\ \mathit{incr}\ \mathit{insT}'
\end{array}
$$

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

$$
\begin{tikzcd}
 & \begin{array}{c}\mathsf{WLHeap} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set} \\ \text{weight-biased leftist heaps}\end{array} \arrow[dl] \arrow[dr] & & \begin{array}{c}\mathsf{LHeap} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set} \\ \text{(rank-biased) leftist heaps}\end{array} \arrow[dl] \arrow[dr] & \\
\begin{array}{c}\mathsf{WLTree} : \mathsf{Nat} \to \mathsf{Set} \\ \text{weight-biased leftist trees}\end{array} \arrow[ddrr] & & \begin{array}{c}\mathsf{Heap} : \mathit{Val} \to \mathsf{Set} \\ \text{heap-ordered trees}\end{array} \arrow[d] & & \begin{array}{c}\mathsf{LTree} : \mathsf{Nat} \to \mathsf{Set} \\ \text{(rank-biased) leftist trees}\end{array} \arrow[ddll] \\
 & & \begin{array}{c}\mathsf{ITree}\ \mathit{Val} : \mathsf{Set} \\ \text{internally labelled trees}\end{array} \arrow[d] & & \\
 & & \begin{array}{c}\mathsf{Tree} : \mathsf{Set} \\ \text{skeletal binary trees}\end{array} & &
\end{tikzcd}
$$

Figure 3.8 Datatypes about leftist heaps and their ornamental relationships.

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. This example is thus a nice demonstration of using the ornament–refinement framework to derive nontrivial types and programs from straightforward markings.

3.4.3 Leftist heaps

Our last 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 **left-**

> **ist property**: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its **right spine** (i.e., the rightmost path from the node in question to an empty node).

From this 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 3.8.

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

$$
\begin{array}{l}
\textbf{data}\ \mathsf{TreeTag}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{'nil}\ \ \ :\ \mathsf{TreeTag} \\
\quad \mathsf{'node}\ :\ \mathsf{TreeTag} \\
\textit{TreeD}\ :\ \mathsf{Desc}\ \top \\
\textit{TreeD}\ \blacksquare\ =\ \sigma\ \mathsf{TreeTag}\ \lambda\ \{\ \mathsf{'nil}\ \mapsto\ \mathsf{v}\ [\,] \\
\qquad\qquad\qquad\qquad\quad ;\ \mathsf{'node}\ \mapsto\ \mathsf{v}\ (\blacksquare :: \blacksquare :: [\,])\ \} \\
\textbf{indexfirst data}\ \mathsf{Tree}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Tree}\ \ni\ \mathsf{nil} \\
\qquad\qquad \mid\ \mathsf{node}\ (t\ :\ \mathsf{Tree})\ (u\ :\ \mathsf{Tree})
\end{array}
$$

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.

(Below we overload $\_\leqslant\_$ to also denote the decidable total ordering on Nat.)

$$
\begin{array}{l}
\mathit{LTreeOD} \;:\; \mathsf{OrnDesc}\ \mathsf{Nat}\ !\ \mathit{TreeD} \\
\mathit{LTreeOD}\ (\mathsf{ok}\ \mathsf{zero}\quad) \;=\; \nabla[\,\text{‘}\mathsf{nil}\,]\ \mathsf{v}\ \blacksquare \\
\mathit{LTreeOD}\ (\mathsf{ok}\ (\mathsf{suc}\ r)) \;=\; \nabla[\,\text{‘}\mathsf{node}\,]\ \Delta[\,l : \mathsf{Nat}\,]\ \Delta[\,r{\leqslant}l : r \leqslant l\,]\ \mathsf{v}\ (\mathsf{ok}\ l\ ,\ \mathsf{ok}\ r\ ,\ \blacksquare)
\end{array}
$$

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{LTree} \;:\; \mathsf{Nat} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Tree}\ \mathsf{zero} \qquad \ni\ \mathsf{nil} \\
\quad \mathsf{Tree}\ (\mathsf{suc}\ r)\ \ni\ \mathsf{node}\ \{l \;:\; \mathsf{Nat}\}\ (r{\leqslant}l \;:\; r \leqslant l)\ (t \;:\; \mathsf{Tree}\ l)\ (u \;:\; \mathsf{Tree}\ r)
\end{array}
$$

Independently, **heap-ordered trees** are also an ornamented version of Tree. The datatype Heap $:$ *Val* $\rightarrow$ 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.

$$
\begin{array}{l}
\mathit{HeapOD} \;:\; \mathsf{OrnDesc}\ \mathit{Val}\ !\ \mathit{TreeD} \\
\mathit{HeapOD}\ (\mathsf{ok}\ b) \;= \\
\quad \sigma\ \mathsf{TreeTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \quad\ \mapsto\ \mathsf{v}\ \blacksquare \\
\qquad\qquad\qquad ;\ \text{‘}\mathsf{node}\ \mapsto\ \Delta[\,x : \mathit{Val}\,]\ \Delta[\,b{\leqslant}x : b \leqslant x\,]\ \mathsf{v}\ (\mathsf{ok}\ x\ ,\ \mathsf{ok}\ x\ ,\ \blacksquare)\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Heap} \;:\; \mathit{Val} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Heap}\ b\ \ni\ \mathsf{nil} \\
\qquad\qquad\ \mid\ \mathsf{node}\ (x \;:\; \mathit{Val})\ (b{\leqslant}x \;:\; b \leqslant x)\ (t \;:\; \mathsf{Heap}\ x)\ (u \;:\; \mathsf{Heap}\ x)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{LHeapOD} \;:\; \mathsf{OrnDesc}\ (!\bowtie!)\ \mathit{pull}\ \mathit{TreeD} \\
\mathit{LHeapOD} \;=\; \lceil \mathit{HeapOD} \rceil \otimes \lceil \mathit{LTreeOD} \rceil
\end{array}
$$

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{LHeap} \;:\; \mathit{Val} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{LHeap}\ b\ \mathsf{zero} \qquad \ni\ \mathsf{nil} \\
\quad \mathsf{LHeap}\ b\ (\mathsf{suc}\ r)\ \ni\ \mathsf{node}\ (x \;:\; \mathit{Val})\ (b{\leqslant}x \;:\; b \leqslant x) \\
\qquad\qquad\qquad\qquad\qquad \{l \;:\; \mathsf{Nat}\}\ (r{\leqslant}l \;:\; r \leqslant l)\ (t \;:\; \mathsf{Heap}\ x\ l)\ (u \;:\; \mathsf{Heap}\ x\ r)
\end{array}
$$

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 $\lceil \mathit{HeapOD} \rceil$,

$$
\begin{array}{l}
\mathsf{Heap}' \;:\; \mathit{Val} \rightarrow \mathsf{Set} \\
\mathsf{Heap}'\; b \;=\; \mathsf{OptP}\; \lceil \mathit{HeapOD} \rceil\; (\mathsf{ok}\; b) \\
\textbf{indexfirst data}\; \mathsf{Heap}' \;:\; \mathit{Val} \rightarrow \mathsf{Tree} \rightarrow \mathsf{Set}\; \textbf{where} \\
\quad \mathsf{Heap}'\; b\; \mathsf{nil} \qquad\qquad \ni\; \mathsf{nil} \\
\quad \mathsf{Heap}'\; b\; (\mathsf{node}\; t\; u) \;\ni\; \mathsf{node}\; (x \;:\; \mathit{Val})\; (b{\leqslant}x \;:\; b \leqslant x) \\
\qquad\qquad\qquad\qquad\qquad\qquad (t' \;:\; \mathsf{Heap}\; x\; t)\; (u' \;:\; \mathsf{Heap}\; x\; u)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{relax} \;:\; \{b\; b' \;:\; \mathit{Val}\} \rightarrow b' \leqslant b \rightarrow \{t \;:\; \mathsf{Tree}\} \rightarrow \mathsf{Heap}'\; b\; t \rightarrow \mathsf{Heap}'\; b'\; t \\
\mathit{relax}\; b'{\leqslant}b\; \{\mathsf{nil} \qquad\;\; \}\; \mathsf{nil} \qquad\qquad\qquad\;\; =\; \mathsf{nil} \\
\mathit{relax}\; b'{\leqslant}b\; \{\mathsf{node}\; \_\; \_\}\; (\mathsf{node}\; x\; b{\leqslant}x\; t\; u) \;=\; \mathsf{node}\; x\; ({\leqslant}\text{-}\mathit{trans}\; b'{\leqslant}b\; b{\leqslant}x)\; t\; u
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{lhrelax} \;:\; \{b\; b' \;:\; \mathit{Val}\} \rightarrow b' \leqslant b \rightarrow \{r \;:\; \mathsf{Nat}\} \rightarrow \mathsf{LHeap}\; b\; r \rightarrow \mathsf{LHeap}\; b'\; r \\
\mathit{lhrelax} \;=\; \mathsf{Upgrade}.u\; \mathit{upg}\; \mathit{id}\; (\lambda\, b'{\leqslant}b\; t \;\mapsto\; \mathit{relax}\; b'{\leqslant}b * \mathit{id}) \\
\quad \textbf{where} \\
\qquad \mathit{ref} \;:\; (b \;:\; \mathit{Val})\; (r \;:\; \mathsf{Nat}) \rightarrow \mathsf{Refinement}\; \mathsf{Tree}\; (\mathsf{LHeap}\; b\; r) \\
\qquad \mathit{ref}\; b\; r \;=\; \mathit{toRefinement} \\
\qquad\qquad\qquad (\otimes\text{-}\mathit{FSwap}\; \lceil \mathit{HeapOD} \rceil\; \lceil \mathit{LTreeOD} \rceil\; \mathit{id}\text{-}\mathit{FSwap}\; \mathit{id}\text{-}\mathit{FSwap} \\
\qquad\qquad\qquad\quad (\mathsf{ok}\; (\mathsf{ok}\; b\; ,\; \mathsf{ok}\; r))) \\
\qquad \mathit{upg} \;:\; \mathsf{Upgrade}
\end{array}
$$

$$
\begin{aligned}
&(\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Tree} \qquad\to \mathsf{Tree} \qquad)\\
&(\{b\ b' : \mathit{Val}\} \to b' \leqslant b \to \{r : \mathsf{Nat}\} \to \mathsf{LHeap}\ b\ r \to \mathsf{LHeap}\ b'\ r)\\
&\mathit{upg}\ =\ \forall^{+}[[\, b : \mathit{Val}\,]]\ \forall^{+}[[\, b' : \mathit{Val}\,]]\ \forall^{+}[\,\_ : b' \leqslant b\,]\\
&\qquad\quad \forall^{+}[[\, r : \mathsf{Nat}\,]]\ \mathit{ref}\ b\ r \rightharpoonup \mathit{toUpgrade}\ (\mathit{ref}\ b'\ r)
\end{aligned}
$$

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 by preorder traversal 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. We hence define the **internally labelled trees** as an ornamentation of skeletal binary trees:

$$
\begin{aligned}
&\mathit{ITreeOD} : \mathsf{Set} \to \mathsf{OrnDesc}\ \top\ !\ \mathit{TreeD}\\
&\mathit{ITreeOD}\ A\ \blacksquare\ =\ \sigma\ \mathsf{TreeTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \ \mapsto\ \mathsf{v}\ \blacksquare\\
&\qquad\qquad\qquad\qquad\qquad\quad ;\ \text{‘}\mathsf{node} \ \mapsto\ \Delta[\,\_ : A\,]\ \mathsf{v}\ (\mathsf{ok}\ \blacksquare\ ,\ \mathsf{ok}\ \blacksquare\ ,\ \blacksquare)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{ITree}\ (A : \mathsf{Set}) : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{ITree}\ A\ \ni\ \mathsf{nil}\\
&\qquad\qquad\quad \mid\ \mathsf{node}\ (x : A)\ (t : \mathsf{ITree}\ A)\ (u : \mathsf{ITree}\ A)
\end{aligned}
$$

on which we can do preorder traversal:

$$
\begin{aligned}
&\mathit{preorder}_{\mathsf{IT}} : \{A : \mathsf{Set}\} \to \mathsf{ITree}\ A \to \mathsf{List}\ A\\
&\mathit{preorder}_{\mathsf{IT}}\ \mathsf{nil} \qquad\qquad\ =\ []\\
&\mathit{preorder}_{\mathsf{IT}}\ (\mathsf{node}\ x\ t\ u)\ =\ x :: \mathit{preorder}_{\mathsf{IT}}\ t \mathbin{+\!\!+} \mathit{preorder}_{\mathsf{IT}}\ u
\end{aligned}
$$

This operation can be upgraded to accept any argument whose type is more informative than $\mathsf{ITree}\ A$. Thus we parametrise the upgraded operation *preorder* by an ornament:

$$
\begin{aligned}
&\mathit{preorder} : \{A\ I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\} \to \mathsf{Orn}\ !\ \lfloor \mathit{ITreeOD}\ A \rfloor\ D \to\\
&\qquad\qquad \{i : I\} \to \mu\ D\ i \to \mathsf{List}\ A\\
&\mathit{preorder}\ \{A\}\ \{I\}\ \{D\}\ O\ =\ \mathsf{Upgrade}.u\ \mathit{upg}\ \mathit{preorder}_{\mathsf{IT}}\ (\lambda\ t\ p \to \blacksquare)\\
&\quad \textbf{where}\ \mathit{upg} : \mathsf{Upgrade}\ (\qquad\qquad\quad \mathsf{ITree}\ A \to \mathsf{List}\ A)\\
&\qquad\qquad\qquad\qquad\qquad\quad (\{i : I\} \to \mu\ D\ i \quad \to \mathsf{List}\ A)\\
&\qquad\quad \mathit{upg}\ =\ \forall^{+}[[\, i : I\,]]\ \mathit{RSem}\ O\ (\mathsf{ok}\ i) \rightharpoonup \mathit{toUpgrade}\ \mathit{idRef}
\end{aligned}
$$

where *idRef* is the identity refinement:

$$
\begin{aligned}
&\mathit{idRef} : \{A : \mathsf{Set}\} \to \mathsf{Refinement}\ A\ A \\
&\mathit{idRef} = \mathbf{record}\ \{\ P = \lambda\ \_ \mapsto \top \\
&\qquad\qquad ;\ i = \mathbf{record}\ \{\ \mathit{to} = \lambda\, a \mapsto (a\ ,\ \blacksquare) \\
&\qquad\qquad\qquad\qquad ;\ \mathit{from} = \lambda\ \{(a\ ,\ \blacksquare) \mapsto a\} \\
&\qquad\qquad\qquad\qquad ;\ \text{proofs of laws}\ \}\ \}
\end{aligned}
$$

There is an ornament from ITree to LHeap, which can be written either directly or by **sequentially composing** the following ornament from ITree to Heap with the ornament *diffOrn-l* $\lceil \mathit{HeapOD} \rceil$ $\lceil \mathit{LTreeOD} \rceil$ from Heap to LHeap:

$$
\begin{aligned}
&\mathit{ITreeD\text{-}HeapD} : \mathsf{Orn}\ !\ \lfloor \mathit{ITreeOD}\ \mathit{Val} \rfloor\ \lfloor \mathit{HeapOD} \rfloor \\
&\mathit{ITreeD\text{-}HeapD}\ (\mathsf{ok}\ b) = \\
&\quad \sigma\ \mathsf{TreeTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \mapsto \mathsf{v}\ [\,] \\
&\qquad\qquad ;\ \text{‘}\mathsf{node} \mapsto \sigma[\,x : \mathit{Val}\,]\ \Delta[\,\_ : b \leqslant x\,]\ \mathsf{v}\ (\mathsf{refl} :: \mathsf{refl} :: [\,])\ \}
\end{aligned}
$$

(Sequential composition of ornaments will be introduced in **??**.) Specialising *preorder* by the ornament gives preorder traversal of a leftist heap.

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 3.9. 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,

```
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 3.9 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 color" [2013]).

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 sub-heap 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.

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:

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

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

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

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

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

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

$$\begin{array}{l}
\mathit{wmerge}\ :\ \{b_0\ :\ \mathit{Val}\}\ \{n_0\ :\ \mathsf{Nat}\} \to \mathsf{WLHeap}\ b_0\ n_0 \to \\
\qquad\qquad\ \{b_1\ :\ \mathit{Val}\}\ \{n_1\ :\ \mathsf{Nat}\} \to \mathsf{WLHeap}\ b_1\ n_1 \to \\
\qquad\qquad\ \{b\ \ :\ \mathit{Val}\} \to b \leqslant b_0 \to b \leqslant b_1 \to \mathsf{WLHeap}\ b\ (n_0 + n_1)
\end{array}$$

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.

3.5 Discussion

Ornaments were first proposed by McBride [2011]. This thesis defines ornaments as relations between descriptions (indexed with an erasure function), and rebrands McBride's ornaments as ornamental descriptions. One obvious advantage of relational ornaments is that they can arise between existing descriptions, whereas ornamental descriptions always produce new descriptions at the more informative end. This makes it possible to complete the commutative square of parallel composition with difference ornaments. Another 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:

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

Between *TwinD 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. Below we sketch a simplistic implementation: 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 indexed by $I \;:\; \mathsf{Set}$ 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

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

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

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

where the coproduct type $\_+\_$ and the join operator $\_\triangledown\_$ are defined as usual:

$$
\begin{aligned}
&\mathbf{data}\ \_+\_\ (A\ B \;:\; \mathsf{Set}) \;:\; \mathsf{Set}\ \mathbf{where}\\
&\quad \mathsf{inl} \;:\; A \to A + B\\
&\quad \mathsf{inr} \;:\; B \to A + B
\end{aligned}
$$

$$
\begin{aligned}
&\_\triangledown\_ \;:\; \{A\ B\ C \;:\; \mathsf{Set}\}\ (A \to C) \to (B \to C) \to A + B \to C\\
&(f \mathbin{\triangledown} g)\ (\mathsf{inl}\ a) \;=\; f\ a\\
&(f \mathbin{\triangledown} g)\ (\mathsf{inr}\ b) \;=\; g\ b
\end{aligned}
$$

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 $\lfloor tD' \rfloor_T$:

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

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

(where $\mathit{idROrn} \;:\; \{I \;:\; \mathsf{Set}\}\ (D \;:\; \mathsf{RDesc}\ I) \to \mathsf{ROrn}\ \mathit{id}\ D\ D$ is the identity response ornament) whose forgetful functions perform suitable injection of constructor tags. Note that the manufactured new description $\lfloor tD \oplus tD' \rfloor_T$ is at the less informative end of *inlOrn* and *inrOrn*. It is thus actually biased to refer to the less informative end of an ornament as "basic", but the examples in this thesis are indeed biased in this sense, being influenced by McBride's original formulation.

Dagand and McBride [2012] later adapted McBride's original ornaments to index-first datatypes, and also proposed "reornaments" as a more efficient representation of promotion predicates, taking full advantage of index-first datatypes. Reornaments are reimplemented in this thesis as optimised predicates using parallel composition, as a result of which we can derive properties about optimised predicates using pullback properties of parallel composition in **??**. Dagand and McBride also extended the notion of ornaments to "functional ornaments", which we generalise to refinements and upgrades. The refinement–upgrade approach is logically clearer and more flexible as it allows us to decouple two constructions:

- ornamental relationship between inductive families, whose refinement semantics gives <u>particular</u> conversion isomorphisms between corresponding types in the inductive families, and
- how conversion isomorphisms <u>in general</u> enable function upgrading, as encoded by the upgrade combinators.

Also, compared to functional ornaments, which are formulated syntactically as a universe and then interpreted to types and operations, upgrades skip syntactic formulation and simply bundle relevant types and operations together, which are then composed semantically by the upgrade combinators. The upgrade mechanism can thus be more easily extended by defining new combinators (which we actually do in **??**). In contrast, had we defined upgrades as a universe, we would have had to employ more complex techniques like data types à la carte [Swierstra, 2008] to gain extensibility. The complexity would not have been justified, because constructing a universe for upgrades in their present form offers no benefit: A universe is helpful only when it is necessary to determine the range of syntactic forms, either for nontrivial computation on the syntactic forms or for facilitation of defining new interpretations of the syntactic forms. Neither is the case with upgrades: we do not need to manipulate the syntactic forms of upgrades, nor do we need to obtain semantic entities other than those captured by the fields of `Upgrade`. In contrast, ornaments do need a universe: we need to know all possible syntactic forms of ornaments in order to compose them in parallel, which cannot be done if all we have are the optimised predicates and ornamental conversion isomorphisms, i.e., the refinement semantics. Indeed, this was what prompted us to go from refinements to ornaments, right before Section 3.2. The universe of ornaments might appear complex, but the complexity is justified by, in particular, the ability to compose ornaments in parallel.

The idea of viewing vectors as promotion predicates was first proposed by Bernardy [2011, page 82], which refers to the realizability transformation defined for pure type systems by Bernardy and Lasson [2011]. The idea is later generalised to "type theory in color" [Bernardy and Guilhem, 2013], which uses modalities inspired by colors in typing to manage relative irrelevance of terms and erasure of irrelevant terms. For simple applications like the ones offered in Section 3.4, type theory in color and ornamentation offer similar approaches, with the former providing more native support for erasure of terms and derivation of promotion predicates. Ornaments, however, are fully computational due to the presence of deletion (∇), which allows arbitrary computations, and can thus specify relationship between datatypes beyond erasure. (**??** will offer a clearer view on the computational power of ornaments.)

It is worth noting that

- constructing functions in coherence with existing ones via upgrades and
- manufacturing internalist operations via externalist composition

are both achieved by extra indexing. For the first case, an upgrade on function types is about constructing a function in coherence with a given one, where coherence is defined (in $\_\rightharpoonup\_$) as mapping related arguments to related results — the coherence property of upgrades is thus comparable to free theorems [Wadler, 1989], but the preserved relation we use in upgrades is the "underlying" relation derived from refinements. To guarantee that a function on more informative types (e.g., a function on lists) is in coherence with a given function on basic types (e.g., a function on natural numbers), we index the more informative types with the underlying value, the results of which are the promotion predicates (e.g., vectors). A promotion proof (e.g., a function on vectors) is then a disguise of the function we wish to implement in the first place, whose type now has extra indexing for enforcing coherence by construction. For the second case, suppose that we are asked to combine the internalist operations $insert_O$ on ordered lists and $insert_V$ on vectors to $insert_{OV}$ on ordered vectors, which involves fusing the ordered list and vector computed by the two operations into an ordered vector as the final result. Not any ordered list and vector can be sensibly fused together, however — they must share the same underlying list for the fusion to make sense. Our solution is to further index the two datatypes with the underlying list, and implement operations on these new datatypes, which are *insert-ordered* and *insert-length*. Now we can easily keep track of the underlying list: the types of the new operations guarantee that, when the input ordered list and vector share the same underlying list, so do the results. Thus the operations can be sensibly combined.

Parallel composition provides logical support for manufacturing composite internalist datatypes, but eventually the central problem is about when and how properties of and operations on actual data structures can be analysed and presented in a meaningful way. Decomposition of a property does not always make sense even when it is logically feasible, and when a decomposition does make sense, it is not the case that the resulting properties should always be treated separately. For example, while it is perfectly logical to analyse red-black trees as internally labelled trees satisfying the red and black properties, the red or black property by itself is useless in practice, and hence it is pointless to develop modules separately for the red and black properties. In contrast, we decomposed the leftist heap property into the leftist property and heap ordering for good reasons: there are operations meaningful for heap-ordered trees without the leftist property, and we can impose different leftist properties on these heap-ordered trees while reusing the operations previously defined for heap-ordered trees. Decomposition of the leftist heap property thus makes sense, but this does not mean that we can treat the leftist property and heap ordering separately all the time — merging of leftist heaps, for example, should be done by considering the leftist property and heap ordering simultaneously, since both properties are essential to the correctness of the merging algorithm — they are not "separable concerns" in this case, in Dijkstra's terminology. Parallel composition is thus merely one small step towards a modular internalist library, since all it provides is logical support of property decomposition, which does not necessarily align with meaningful separation of concerns. It requires further consideration to reorganise data structures and algorithms — together with the various properties they satisfy, which are now first-class entities — in a way that makes proper use of the new logical support.

Bibliography

Jean-Philippe Bernardy [2011]. *A Theory of Parametric Polymorphism and an Application*. Ph.D. thesis, Chalmers University of Technology. ↰ page 61

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 55 and 61

Jean-Philippe Bernardy and Mark Lasson [2011]. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computation Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag. doi: 10.1007/978-3-642-19805-2_8. ↰ page 61

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. ↰ pages 22 and 58

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 59 and 60

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ pages 58 and 59

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 21

Chris OKASAKI [1999]. *Purely functional data structures*. Cambridge University Press. ↰ pages 37, 41, 49, and 54

Wouter SWIERSTRA [2008]. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. doi: 10.1017/S0956796808006758. ↰ pages 58 and 60

Philip WADLER [1989]. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM. ↰ page 61

Todo list