Analysis and synthesis of inductive families

Hsiang-Shang Ko

17 October 2013

Contents

1 Introduction **1**

2 From intuitionistic type theory to dependently typed programming **3**

3 Refinements and ornaments **5**

3.1 Refinements 5

3.1.1 Refinements between individual types 5

3.1.2 Upgrades 7

3.1.3 Refinement families 10

3.2 Datatype descriptions 10

3.3 Ornaments 11

3.4 Two examples about heaps 13

3.4.1 Binomial heaps 13

3.4.2 Leftist heaps 20

3.5 Discussion 26

4 Categorical organisation of the ornament–refinement framework **27**

5 Relational algebraic ornaments **29**

6 Categorical equivalence of ornaments and relational algebras 33

7 Conclusion 35

7.1 Future work 35

Chapter 1

Introduction

"datatypes" for inductive families

Chapter 2

From intuitionistic type theory to dependently typed programming

We start with an introduction to intuitionistic type theory [Martin-Löf, 1984] and dependently typed programming [Altenkirch et al., 2005; McBride, 2004] using the Agda language [Norell, 2007, 2009; Bove and Dybjer, 2009]. Intuitionistic type theory was developed by Martin-Löf to serve as a foundation of intuitionistic mathematics like Bishop's renowned work on constructive analysis [Bishop and Bridges, 1985]. While originated from intuitionistic type theory, dependently typed programming is more concerned with mechanisation and practicalities, and is influenced by the program construction movement. It has thus departed from the mathematical traditions considerably, and deviations can be found from syntactic presentations to the underlying philosophy.

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

Chapter 3

Refinements and ornaments

the second half of the insertion example, i.e., solution (the first half is in Chapter 2)

3.1 Refinements

3.1.1 Refinements between individual types

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

universe polymorphism

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

In such a refinement, Y is usually a more informative variant of the basic type X: the elements of Y exactly correspond to the elements of X paired with proofs that the promotion predicate P is satisfied. Computationally, any element of Y can be analysed (by $\mathsf{Iso}.to\ i$) into an underlying element $x : X$

and a proof that x satisfies the promotion predicate P (which we will call a *promotion proof* for x), and conversely, if a basic element $x : X$ satisfies P, then it can be promoted (by $\mathsf{Iso}.\mathit{from}\ i$) to an element of Y. We denote the forgetful computation of the underlying element, i.e., $\mathsf{outl} \circ \mathsf{Iso}.\mathit{to}\ i$, as $\mathsf{Refinement}.\mathit{forget}$.

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

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

An ordered list of type $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ a$ can be analysed into a list $xs : \mathsf{List}\ A$ and a proof of type $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a\ xs$ that the list xs is ordered and bounded below by a; conversely, a list satisfying $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a$ can be promoted to an ordered list of type $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ a$. (*End of example.*)

Example (*refinement from natural numbers to lists*). Let $A : \mathsf{Set}$. 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 promotion isomorphism of type

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

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*: a basic element can have more than one promotion proofs, and consequently the more informative type in a refinement can have more elements than the basic type does. (*End of example.*)

canonical refinements and the residual/difference view

3.1.2 Upgrades

When we move on to function types, however, refinements are less useful. Even when we have extensional equality for functions so isomorphisms between function types make sense, the requirement that a promotion isomorphism exists is still too strong. For example, it is not — and should not be — possible to have a refinement from $\mathsf{Nat} \to \mathsf{Nat}$ to $\mathsf{List\ Nat} \to \mathsf{List\ Nat}$, despite that the component types Nat and List Nat are related by a refinement: If such a refinement existed, we would be able to extract from any function $f : \mathsf{List\ Nat} \to \mathsf{List\ Nat}$ 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.

Comparison (*Type Theory in Colour*).

Bernardy and Guilhem [2013]

(*End of comparison.*)

It is only the analytic 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 a function of type $\mathsf{Nat} \to \mathsf{Nat}$ to a function of type $\mathsf{List\ Nat} \to \mathsf{List\ Nat}$, we can use

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

as the promotion predicate: Consider the refinement from Nat to List Nat. Given a function $f : \mathsf{Nat} \to \mathsf{Nat}$ and a promotion proof $q : Q\,f$, we can synthesise a function $f' : \mathsf{List\ Nat} \to \mathsf{List\ Nat}$ by

$$f' = \mathsf{Iso}.\mathit{from}\ i \circ (f * q) \circ \mathsf{Iso}.\mathit{to}\ i$$

which satisfies the *coherence property* [Dagand and McBride, 2012]

Explain the meaning of this (scoping).

definition of $*$

diagram

$$f \circ \mathit{forget} \doteq \mathit{forget} \circ f'$$

What is "recursive structure"?

i.e., f' processes the recursive structure of its input in the same way as f does.

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

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

to be made more precise

We will not write upgrades by hand; rather, we will define several combinators for synthesising upgrades.

Upgrades from refinements

Intuitively, upgrades amount to the promoting direction of refinements. We should thus be able to extract upgrades from refinements. Comparing the definitions of refinements and upgrades, we see that, to turn a refinement into an upgrade, only the coherence property and the coherence proof are missing — the promotion predicate and the upgrading operation are readily obtainable. The most important question we should ask is thus: with respect to a refinement from X to Y, what is a sensible coherence property on X and Y? In other words, when should we say that a basic element $x : X$ and a promoted element $y : Y$ are in coherence? The fundamental idea of a refinement is that within any promoted element $y : Y$ there is an underlying basic element $\mathit{forget}\ y : X$, so it should be sensible to define that $x : X$ is in coherence with y

when $\mathit{forget}\ y \equiv x$. In fact, we can prove that the promotion predicate of any refinement from X to Y is pointwise isomorphic to the *canonical* promotion predicate $\lambda\, x \mapsto \Sigma\langle y : Y \rangle\ \mathit{forget}\ y \equiv x$:

$$\begin{aligned}&\mathit{coherence} : \{X\ Y : \mathsf{Set}\}\ (r : \mathsf{Refinement}\ X\ Y) \to \\ &\qquad (x : X) \to \mathsf{Refinement}.P\ r\ x \cong \Sigma\langle y : Y \rangle\ \mathsf{Refinement}.\mathit{forget}\ r\ y \equiv x\end{aligned}$$

That is, a promotion proof for $x : X$ always amounts to saying that there exists a promoted element whose underlying element is x. We can now extract an upgrade from a refinement by setting the coherence property as $\lambda\, x\ y \mapsto \mathit{forget}\ y \equiv x$ and producing the coherence proof by *coherence*.

redundant?

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

Proof of *coherence* (canonical refinements, etc), which should perhaps be moved to the section on refinements.

Remark (*Upgrade-style refinements*). (*End of remark.*)

Composition of upgrades

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

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

Note that there should be a *refinement* between the source types X and Y, rather than just an upgrade. Consequently, we can produce upgrades between curried multi-argument function types but not between higher-order function types. As for the reason, we need to look at the detail of the combinator: Let $r : \mathsf{Refinement}\ X\ Y$ and $s : \mathsf{Upgrade}\ Z\ W$.

```
_⇀_ : {X Y Z W : Set} →
      Refinement X Y → Upgrade Z W → Upgrade (X → Z) (Y → W)
r ⇀ s = record
  { P = λ f ↦ ∀ x → Refinement.P r x → Upgrade.P s (f x)
  ; C = λ f g ↦ ∀ x y → Upgrade.C (toUpgrade r) x      y      →
                        Upgrade.C s                  (f x) (g y)
  ; u = λ f h ↦ Upgrade.u s _ ∘ uncurry h ∘ Iso.to (Refinement.i r)
  ; c = λ { f h ._ y refl ↦ let xp = (Iso.to (Refinement.i r) y)
                             in Upgrade.c s (f (outl xp)) (uncurry h xp) } }
```

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

```
r = Nat-List Nat : Refinement Nat (List Nat)
```

and the upgrade extracted from r, we get an upgrade

```
r ⇀ toUpgrade r : Upgrade (Nat → Nat) (List Nat → List Nat)
```

(*End of example.*)

More combinators.

Comparison (*functional ornaments*).

Dagand and McBride [2012], origin of coherence property, no need to construct a universe

(*End of comparison.*)

3.1.3 Refinement families

3.2 Datatype descriptions

```
Desc : Set → Set₁
μ : {I : Set} → Desc I → (I → Set)
data RDesc (I : Set) : Set₁ where
```

$$
\begin{aligned}
&\mathsf{v} \;:\; (is \;:\; \mathsf{List}\; I) \to \mathsf{RDesc}\; I \\
&\sigma \;:\; (S \;:\; \mathsf{Set})\; (D \;:\; S \to \mathsf{RDesc}\; I) \to \mathsf{RDesc}\; I \\
&[\![\_]\!] \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{RDesc}\; I \to (I \to \mathsf{Set}) \to \mathsf{Set} \\
&[\![\; \mathsf{v}\; is \;\;]\!]\; X \;=\; \mathbb{M}\; is\; X \\
&[\![\; \sigma\; S\; D \;]\!]\; X \;=\; \Sigma \langle\, s : S \,\rangle\;\; [\![\; D\; s \;]\!]\; X \\
&\mathsf{Desc} \;:\; \mathsf{Set} \to \mathsf{Set}_1 \\
&\mathsf{Desc}\; I \;=\; I \to \mathsf{RDesc}\; I \\
&\mathbb{F} \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{Desc}\; I \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \\
&\mathbb{F}\; D\; X\; i \;=\; [\![\; D\; i \;]\!]\; X \\
&\textbf{data}\; \mu\; \{I \;:\; \mathsf{Set}\}\; (D \;:\; \mathsf{Desc}\; I) \;:\; I \to \mathsf{Set}\; \textbf{where} \\
&\quad \mathsf{con} \;:\; \mathbb{F}\; D\; (\mu\; D) \rightrightarrows \mu\; D \\
&\mathbb{M} \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{List}\; I \to (I \to \mathsf{Set}) \to \mathsf{Set} \\
&\mathbb{M}\; [\,]\;\;\;\;\;\; X \;=\; \top \\
&\mathbb{M}\; (i :: is)\; X \;=\; X\; i \times \mathbb{M}\; is\; X
\end{aligned}
$$

first-order vs higher-order representation: change the type of v to $(S \;\; : \; \mathsf{Set}) \to (S \to I) \to \mathsf{RDesc}\; I$ and define $\mathbb{M}\; f\; X \;\; = \;\; (s \;\; : \;\; S) \to X\; (f\;\; s)$: $\{I\;\; S \;\; : \;\; \mathsf{Set}\} \to (S \to I) \to (I \to \mathsf{Set}) \to \mathsf{Set}$; $\mathsf{List}\;\; I$ is extensionally isomorphic to $\Sigma \langle\, n : \mathsf{Nat} \,\rangle\;\; \mathsf{Fin}\; n \to I$

3.3 Ornaments

Think of the appearance of index-first datatype declarations as an informal hint that syntactic sugar will be used.

Evolutionary remark (*ornaments as relations*). We define ornaments as relations between descriptions (indexed by an erasure function), whereas the original ornaments [McBride, 2011; Dagand and McBride, 2012] are rebranded as ornamental descriptions. One obvious advantage of relational ornaments is that they can arise between *existing* descriptions, whereas ornamental descriptions always produce (definitionally) new descriptions at the more informative

end. This also means that there can be multiple ornaments between a pair of descriptions. For example, consider the datatype

```
indexfirst data Square (A : Set) : Set where
  Square A ∋ _,_ (x : A) (y : A)
```

Between the description of Square A and itself, we have the identity ornament

$$\sigma\langle x : A\rangle\ \sigma\langle y : A\rangle\ \mathsf{v}\ [\,]$$

and the ornament

$$\Delta\langle x : A\rangle\ \Delta\langle y : A\rangle\ \nabla\langle y\rangle\ \nabla\langle x\rangle\ \mathsf{v}\ [\,]$$

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

```
TDesc : Set → Set₁
TDesc I = Σ⟨ C : I → Set ⟩ ((i : I) → C i → RDesc I)
```

Tagged descriptions are decoded to ordinary descriptions by

```
⌊_⌋T : {I : Set} → TDesc I → Desc I
⌊C , D⌋T i = σ (C i) (D i)
```

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

coproduct-related definitions

```
_⊕_ : {I : Set} → TDesc I → TDesc I → TDesc I
(C , D) ⊕ (C′ , D′) = (λ i ↦ C i + C′ i) , (λ i ↦ D i ▽ D′ i)
```

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

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

whose forgetful functions perform suitable injection of constructor tags. Note that the synthesised new description $\lfloor\, tD \oplus tD' \,\rfloor_T$ is at the less informative end of *inlOrn* and *inrOrn*. (*End of evolutionary remark.*)

Example?

3.4 Two examples about heaps

Both examples are adapted from Okasaki's work [1999]. The first example about *binomial heaps* shows that Okasaki's idea of *numerical representations* can be elegantly captured by ornaments and the coherence property of upgrades, and the second example about *leftist heaps* demonstrates the power of parallel composition of ornaments by treating heap ordering and leftist balancing properties modularly.

3.4.1 Binomial heaps

We are 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 value 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* is a binary number in which every 1-digit stores a *binomial tree* — the actual place for storing elements — whose size is exactly the weight of the digit. The number of elements stored in a binomial heap is therefore exactly the value of the underlying binary number. Merging two binomial heaps is analogous to addition of two binary numbers, with carrying corresponding to combining two binomial trees of the same size. Okasaki thus proposed

point to a figure

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 binomial heaps as an example, which is presented in detail below.

Binomial trees

The basic building blocks of binomial heaps are *binomial trees*, in which elements are stored. We assume the type of elements to be *Val*, which is equipped with a decidable total ordering. Binomial trees are defined inductively on their *rank*, which is a natural number:

- a binomial tree of rank 0 is a single node storing an element, 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 immediately deduce that a binomial tree of rank r has exactly 2^r elements. To actually define binomial trees as a datatype, however, an alternative definition 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. 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

picture

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:

```
BTreeD : Desc Nat
BTreeD r = σ⟨ _ : Val ⟩ ∨ (descend r)

BTree : Nat → Set
BTree = μ BTreeD
```

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

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

Remark (*raw, sugar-free binomial trees*). In *BTreeD*, we are exploiting the full computational power of Desc, computing the list of recursive indices from the requested index. 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. (*End of remark*.)

Here we define several operations on binomial trees. Any binomial tree starts with a root element, which can be extracted by

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

Given two binomial trees of the same rank r, one can be attached under the root of the other, forming a single binomial tree of rank $1 + r$.

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

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

```
link : {r : Nat} → BTree r → BTree r → BTree (suc r)
link t u with root t ⩽? root u
```

```
link t u | yes _ = attach u t
link t u | no  _ = attach t u
```

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, however).

From binary numbers to binomial heaps

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

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

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

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

The intended interpretation of binary numbers is given by

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

That is, the digits in a binary number of type Bin are ordered from the least significant digit to the most significant one, 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, and can be expressed straightforwardly as an ornamentation of binary numbers. To ensure that the binomial trees in binomial heaps have the right rank, the datatype BHeap : Nat → Set is indexed with a "starting rank": if a binomial heap of type BHeap r is nonempty (i.e., not nil), then its first digit has rank r (and stores a binomial tree of rank r if the digit is one), and the rest of the heap is indexed with rank $1 + r$.

```
BHeapOD : OrnDesc Nat
BHeapOD (ok r) = σ BinTag λ { 'nil  ↦ v tt
                            ; 'zero ↦ v (ok (suc r), tt)
                            ; 'one  ↦ Δ⟨ t : BTree r ⟩ v (ok (suc r), tt) }

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

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

Increment and insertion, in coherence

Increment of binary numbers is defined by

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

The corresponding operation on binomial heaps is insertion of a binomial tree into a binomial heap (of matching rank):

```
insT : {r : Nat} → BTree r → BHeap r → BHeap r
insT t nil      = one t nil
```

$$\begin{aligned}
&\mathit{insT}\ t\ (\mathsf{zero}\quad h) &&= \mathsf{one}\ t\ h\\
&\mathit{insT}\ t\ (\mathsf{one}\ u\ h) &&= \mathsf{zero}\ (\mathit{insT}\ (\mathit{link}\ t\ u)\ h)
\end{aligned}$$

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

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

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

$$\begin{aligned}
&\mathit{upg} : \mathsf{Upgrade}\ (\mathsf{Bin} \to \mathsf{Bin})\ (\{r : \mathsf{Nat}\} \to \mathsf{BTree}\ r \to \mathsf{BHeap}\ r \to \mathsf{BHeap}\ r)\\
&\mathit{upg}\ =\ \forall^+\langle\!\langle\, r : \mathsf{Nat}\,\rangle\!\rangle\ \forall^+\langle\, _ : \mathsf{BTree}\ r\,\rangle\\
&\qquad\quad \mathbf{let}\ \mathit{ref} : \mathsf{Refinement\ Bin}\ (\mathsf{BHeap}\ r)\\
&\qquad\qquad\quad \mathit{ref}\ =\ \mathit{RSem}'\ \lceil \mathit{BHeapOD} \rceil\ (\mathsf{ok}\ r)\\
&\qquad\quad \mathbf{in}\ \mathit{ref} \rightharpoonup \mathit{toUpgrade\ ref}
\end{aligned}$$

The upgrade *upg* says that, relative to the type of *incr*, the type of *insT* has two new arguments — the implicit argument $r : \mathsf{Nat}$ and the explicit argument of type $\mathsf{BTree}\ r$ — and that the two occurrences of $\mathsf{BHeap}\ r$ in the type of *insT* refine the corresponding occurrences of Bin in the type of *incr* using the refinement semantics of the ornament from Bin to $\mathsf{BHeap}\ r$. The coherence property $\mathsf{Upgrade}.C\ \mathit{upg}\ \mathit{incr}\ \mathit{insT}$ (i.e., *incr* and *insT* are in coherence with respect to *upg*) expands to

$$\{r : \mathsf{Nat}\}\ (t : \mathsf{BTree}\ r)\ (b : \mathsf{Bin})\ (h : \mathsf{BHeap}\ r) \to$$

$$\mathit{forget}\ \lceil \mathit{BHeapOD} \rceil\ h \equiv b \to \mathit{forget}\ \lceil \mathit{BHeapOD} \rceil\ (\mathit{insT}\ t\ h) \equiv \mathit{incr}\ b$$

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

We have seen that the coherence property of *upg* is appropriate for characterising the resemblance of *incr* and *insT*; proving that it holds for *incr* and *insT* is a separate matter, though. We can, however, avoid doing a separate proof by writing insertion with a more precise type *in the first place* such that, from this more precisely typed version, we can derive insertion with the original type which satisfies the coherence property automatically. The above process is fully supported by the mechanism of upgrades. In particular, the more precise type for insertion is given by the promotion predicate of *upg* (applied to *incr*), and the more precisely typed version of insertion acts as a promotion proof of *incr* with respect to *upg*.

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

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

Here a more helpful interpretation is that BHeap′ is a datatype of binomial heaps additionally indexed with the underlying binary number. The type Upgrade.P *upg 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* does. Insertion can now be implemented as

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

$$
\begin{aligned}
&\mathit{insT'}\ t\ \mathsf{nil}\ \ \ \ \ \ \mathsf{nil}\ \ \ \ \ \ \ \ &&= \mathsf{one}\ t\ \mathsf{nil}\\
&\mathit{insT'}\ t\ (\mathsf{zero}\ b)\ (\mathsf{zero}\ \ h) &&= \mathsf{one}\ t\ h\\
&\mathit{insT'}\ t\ (\mathsf{one}\ \ b)\ (\mathsf{one}\ u\ h) &&= \mathsf{zero}\ (\mathit{insT'}\ (\mathit{link}\ t\ u)\ h)
\end{aligned}
$$

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

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

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

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

which is automatically in coherence with *incr*:

$$
\begin{aligned}
&\mathit{coherence\text{-}property}\ :\\
&\quad \{r\ :\ \mathsf{Nat}\}\ (t\ :\ \mathsf{BTree}\ r)\ (b\ :\ \mathsf{Bin})\ (h\ :\ \mathsf{BHeap}\ r) \to\\
&\quad \mathit{forget}\ \lceil \mathit{BHeapOD} \rceil\ h\ \equiv\ b \to \mathit{forget}\ \lceil \mathit{BHeapOD} \rceil\ (\mathit{insT}\ t\ h)\ \equiv\ \mathit{incr}\ b\\
&\mathit{coherence\text{-}property}\ =\ \mathsf{Upgrade}.c\ \mathit{upg}\ \mathit{incr}\ \mathit{insT'}
\end{aligned}
$$

3.4.2 Leftist heaps

Clean up in progress.

In Okasaki's words:

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

From this passage we can immediately decompose the concept of leftist heaps into three: leftist heaps (i) are binary trees that (ii) are heap-ordered and (iii) satisfy the leftist property. This suggests that there is a basic datatype of binary trees together with two ornamentations, one expressing heap ordering and the other the leftist property. After defining a datatype of constructor tags,

```
data TreeTag : Set where
  'tip  : TreeTag
  'fork : TreeTag
```

the datatype of binary trees is

```
indexfirst data Tree : Set where
  Tree ∋ tip
       | fork (t : Tree) (u : Tree)
TreeD : Desc ⊤
TreeD _ = σ TreeTag λ { 'tip  ↦ v []
                      ; 'fork ↦ v (tt :: tt :: []) }
```

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

```
indexfirst data LTree : Nat → Set where
  Tree zero    ∋ tip
  Tree (suc r) ∋ fork (l : Nat) (r≤l : r ≤ l) (t : Tree l) (u : Tree r)
LTreeOD : OrnDesc Nat ! TreeD
LTreeOD (ok zero   ) = ∇⟨ 'tip ⟩ v tt
LTreeOD (ok (suc r)) = ∇⟨ 'fork ⟩ Δ⟨ l : Nat ⟩ Δ⟨ r≤l : r ≤ l ⟩ v (ok l, ok r, tt)
```

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

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

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

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

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

Display the predicate(?)

```
relax : {b b′ : Val} → b′ ⩽ b →
        {t : Tree} → OptP ⌈ HeapO ⌉ (ok b) t → OptP ⌈ HeapO ⌉ (ok b′) t
relax b′⩽b {tip}        _                  = con tt
relax b′⩽b {fork _ _} (con (x , b⩽x , t , u)) = con (x , ⩽-trans b′⩽b b⩽x , t , u)
```

(where $\leqslant$-*trans* is transitivity of $\_\leqslant\_$) then we can lift it so as to modify only the heap-ordering portion of a leftist heap:

Revise to use upgrades.

```
lhrelax : ∀ {b b′} → b′ ⩽ b → ∀ {r} → LHeap b r → LHeap b′ r
lhrelax {b} {b′} b′⩽b {r} =
  Iso.from (Refinement.ℜ re (ok (ok b′ , ok r))) ∘
```

```
    (id ** (relax b'≤b ** id)) ∘ Iso.to (Refinement.ℜ re (ok (ok b, ok r)))
  where
    re : Refinement (μ TreeD) (μ LHeapD)
    re = toRefinement (Swap −⊗ ⌈HeapO⌉ ⌈LTreeO⌉ idSwap idSwap)
```

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

```
indexfirst data ITree (A : Set) : Set where
  ITree A ∋ tip
          | fork (x : A) (t : ITree A) (u : ITree A)

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

on which we can do pre-order traversal:

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

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

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

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

```
toList : {b : Val} {r : Nat} → LHeap b r → List Val
```

$$\mathit{toList} \;=\; \mathit{preorder} \circ \mathit{forget}\ (\mathit{ITreeD\text{-}HeapD} \odot \mathit{diffOrn\text{-}l}\ \lceil \mathit{HeapOD} \rceil\ \lceil \mathit{LTreeOD} \rceil)$$

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

$$\begin{aligned}\mathit{merge} \;:\; &\{b \;:\; \mathit{Val}\}\ \{r \;:\; \mathsf{Nat}\} \to \mathsf{LHeap}\ b\ \ r \;\to \\ &\{b' \;:\; \mathit{Val}\}\ \{r' \;:\; \mathsf{Nat}\} \to \mathsf{LHeap}\ b'\ r' \to \Sigma\langle\, r'' : \mathsf{Nat} \,\rangle\ \ \mathsf{LHeap}\ (b \sqcap b')\ r''\end{aligned}$$

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 3.1. It is a more precisely typed version of Okasaki's implementation, split into two mutually recursive functions to make the two-level induction clear to Agda's termination checker, and conversions are added to establish the correct bounds.

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

$$\begin{aligned}&\textbf{indexfirst data}\ \mathsf{WLTree}\ :\ \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{WLTree\ zero} \quad\ni\ \mathsf{tip}\\ &\quad \mathsf{WLTree}\ (\mathsf{suc}\ n)\ \ni\ \mathsf{fork}\ (l\ :\ \mathsf{Nat})\ (r\ :\ \mathsf{Nat})\\ &\qquad\qquad\qquad\qquad (r{\leqslant}l\ :\ r \leqslant l)\ (n{\equiv}l{+}r\ :\ n \equiv l + r)\\ &\qquad\qquad\qquad\qquad (t\ :\ \mathsf{WLTree}\ l)\ (u\ :\ \mathsf{WLTree}\ r)\\ &\mathit{WLTreeOD}\ :\ \mathsf{OrnDesc\ Nat}\ !\ \mathit{TreeD}\\ &\mathit{WLTreeOD}\ (\mathsf{ok\ zero}\quad) \;=\; \nabla\langle\, \text{`}\mathsf{tip} \,\rangle\ \mathsf{v\ tt}\\ &\mathit{WLTreeOD}\ (\mathsf{ok}\ (\mathsf{suc}\ n)) \;=\; \nabla\langle\, \text{`}\mathsf{fork} \,\rangle\ \ \Delta\langle\, l : \mathsf{Nat} \,\rangle\ \ \Delta\langle\, r : \mathsf{Nat} \,\rangle\end{aligned}$$

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

Figure 3.1: Merging two leftist heaps.

$$\Delta\langle \_ : r \leqslant l \rangle \;\; \Delta\langle \_ : n \equiv l + r \rangle \;\; \mathsf{v} \; (\mathsf{ok}\; l, \mathsf{ok}\; r, \mathsf{tt})$$

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

$$\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{WLHeap} : \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
&\quad \mathsf{WLHeap}\ b\ \mathsf{zero} \quad \ni \quad \mathsf{tip} \\
&\quad \mathsf{WLHeap}\ b\ (\mathsf{suc}\ n) \ni \mathsf{fork}\ (x : \mathit{Val})\ (b{\leqslant}x : b \leqslant x) \\
&\qquad\qquad\qquad\qquad (l : \mathsf{Nat})\ (r : \mathsf{Nat}) \\
&\qquad\qquad\qquad\qquad (r{\leqslant}l : r \leqslant l)\ (n{\equiv}l{+}r : n \equiv l + r) \\
&\qquad\qquad\qquad\qquad (t : \mathsf{WLHeap}\ x\ l)\ (u : \mathsf{WLHeap}\ x\ r) \\
&\mathit{WLHeapD} : \mathsf{Desc}\ (! \bowtie !) \\
&\mathit{WLHeapD} = \lfloor \lceil \mathit{HeapOD} \rceil \otimes \lceil \mathit{WLTreeOD} \rceil \rfloor
\end{aligned}$$

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

3.5 Discussion

Why ornaments?

Chapter 4

Categorical organisation of the ornament–refinement framework

Chapter 5

Relational algebraic ornaments

The three datatypes Nat, List A, and Vec A are evidently related: a list is a natural number whose cons nodes are decorated with elements of A, and a vector is a list enriched with length information. Such relationship can be seen by "overlaying" one datatype declaration on the other: for example, the declaration of List A differs from that of Nat only in an extra field $(a : A)$ in the cons constructor, and the declaration of Vec A differs from that of List A in that (i) the index set is changed from $\top$ to Nat, (ii) the cons constructor has two extra fields, and (iii) the index of the recursive position is specified to be m. Such differences between datatype declarations are encoded as *ornaments*. Whenever there is an ornament between two datatypes, there is a forgetful function from the more informative datatype to the other, erasing information according to the ornament's specification of datatype differences. For example, we have a forgetful function from lists to natural numbers that discards elements associated with cons nodes — i.e., it computes the length of a list — and another one from vectors to lists which removes all length information from a vector and returns the underlying list.

Ornaments constitute the second underlying universe:

$$\mathsf{Orn} : \{I\ J : \mathsf{Set}\}\ (e : J \to I)\ (D : \mathsf{Desc}\ I)\ (E : \mathsf{Desc}\ J) \to \mathsf{Set}_1$$

An ornament $O : \mathsf{Orn}\ e\ D\ E$ specifies the difference between the more informative description E and the basic description D, and is parametrised by an "index erasure" function e from the index set of E to that of D. The ornament gives rise to a forgetful function

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

For example, there are families of ornaments

$$\mathit{NatD} - \mathit{ListD} : (A : \mathsf{Set}) \to \mathsf{Orn}\ !\ \mathit{NatD}\ (\mathit{ListD}\ A)$$

and

$$\mathit{ListD} - \mathit{VecD} : (A : \mathsf{Set}) \to \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ (\mathit{VecD}\ A)$$

(where $! = \mathit{const}\ \mathsf{tt}$) that encode the differences between the list-like datatypes. The function

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

computes the length of a list, and the function

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

computes the underlying list of a vector.

Ornamental descriptions. Ornaments arise between existing datatype descriptions. The typical scenario of using ornaments, however, is first modifying a base description into a more informative one and then specifying an ornament between the two descriptions. *Ornamental descriptions* are introduced to combine the two steps into one:

$$\mathsf{OrnDesc} : \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I)\ (D : \mathsf{Desc}\ I) \to \mathsf{Set}_1$$

An ornamental description

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

is like a new description of type Desc J, but is written relative to a base description D such that not only can we extract the new description

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

but we can also extract an ornament from the base description D to the new description

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

An ornamental description is a convenient way to specify a new datatype that has an ornamental relationship with an existing one; it might be thought of as simultaneously denoting the new description and the ornament — the floor and ceiling brackets $\lfloor \_ \rfloor$ and $\lceil \_ \rceil$ are added to resolve ambiguity.

Example. Let $\_\leqslant_{A\_} \ : \ A \to A \to \mathsf{Set}$ be an ordering on A and declare a datatype of ordered lists (parametrised by A and $\_\leqslant_{A\_}$) indexed by a lower bound under this ordering:

indexfirst data OrdList $A\ \_\leqslant_{A\_} \ : \ A \to \mathsf{Set}$ **where**
  OrdList $A\ \_\leqslant_{A\_}\ b$
    accepts nil
      | cons $(a \ : \ A)\ (leq \ : \ b \leqslant A\ a)\ (as \ : \ \mathsf{OrdList}\ A\ \_\leqslant_{A\_}\ a)$

This datatype can be thought of as being decoded from an ornamental description

$$OrdListOD\ A\ \_\leqslant_{A\_} \ : \ \mathsf{OrnDesc}\ A\ !\ (ListD\ A)$$

which inserts the field *leq* and refines the index of the recursive position to a. That is, the underlying description for OrdList is

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

(so OrdList $A\ \_\leqslant_{A\_}\ b$ desugars to $\mu\ \lfloor OrdListOD\ A\ \_\leqslant_{A\_} \rfloor\ b$), and

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

is the ornament from lists to ordered lists.

Chapter 6

Categorical equivalence of ornaments and relational algebras

algebras corresponding to singleton ornaments and ornaments for optimised predicates; banana-split law corresponding to parallel composition; optimised predicates for functional algebraic ornaments amount to equality

Chapter 7

Conclusion

7.1 Future work

- functor-level abstraction

Bibliography

Thorsten Altenkirch, Conor McBride, and James McKinna [2005]. Why dependent types matter. Available at `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`.

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.

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag.

Ana Bove and Peter Dybjer [2009]. Dependent types at work. In *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer-Verlag.

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.

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.

Per Martin-Löf [1984]. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170.

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*.

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

Ulf Norell [2009]. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag.

Chris Okasaki [1999]. *Purely functional data structures*. Cambridge University Press.

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

Todo list

"datatypes" for inductive families 1

the second half of the insertion example, i.e., solution (the first half is in Chapter 2) 5

universe polymorphism 5

canonical refinements and the residual/difference view 6

Bernardy and Guilhem [2013] 7

Explain the meaning of this (scoping). 7

definition of $*$ 7

diagram 7

What is "recursive structure"? 8

to be made more precise 8

redundant? 9

Proof of *coherence* (canonical refinements, etc), which should perhaps be moved to the section on refinements. 9

More combinators. 10

Dagand and McBride [2012], origin of coherence property, no need to construct a universe 10

first-order vs higher-order representation: change the type of v to $(S : \mathsf{Set}) \to (S \to I) \to \mathsf{RDesc}\ I$ and define $\mathbb{M}\, f\ X = (s : S) \to X\ (f\ s)$: $\{I\ S : \mathsf{Set}\} \to (S \to I) \to (I \to \mathsf{Set}) \to \mathsf{Set}$; List I is extensionally isomorphic to $\Sigma \langle\, n : \mathsf{Nat} \,\rangle\ \mathsf{Fin}\ n \to I$ 11

Think of the appearance of index-first datatype declarations as an informal hint that syntactic sugar will be used. 11

coproduct-related definitions 12

Example? 13

point to a figure 13

picture 14

Clean up in progress. 20

Display the predicate(?) 22

Revise to use upgrades. 22

Why ornaments? 26

algebras corresponding to singleton ornaments and ornaments for optimised predicates; banana-split law corresponding to parallel composition; optimised predicates for functional algebraic ornaments amount to equality 33