Analysis and synthesis of inductive families

Hsiang-Shang Ko

14 January 2014

Contents

Chapter 1

Introduction

program correctness by construction (from specifications to programs); type theory (unification of logic and computation and/vs types as classification/specification); new direction of program derivation, while inheriting problems

"datatypes" for inductive families

Chapter 2

From intuitionistic type theory to dependently typed programming

specific issues regarding practical programming with type theory (including introduction to Agda); lead into internalism vs externalism

We start with an introduction to intuitionistic type theory [Martin-Löf, 1984b] 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-correctness-by-construction movement. It has thus departed from the mathematical traditions considerably, and deviations can be found from syntactic presentations to the underlying philosophy.

2.1 Notion of computation in type theory

Mathematics is all about mental constructions, that is, the intuitive grasp and manipulation of mental objects, the intuitionists say [Dummett, 2000; Heyting, 1971]. Take the natural numbers as an example. We have a distinct idea of how natural numbers are built: start from an origin 0, and form its successor 1, and then the successor of 1, which is 2, and so on. In other words, it is in our nature to be able to count, and counting is just the way the natural numbers are constructed. This construction then gives a specification of when we can immediately recognise a natural number, namely when it is 0 or a successor of some other natural number, and this specification of immediately recognisable forms is one of the conditions of forming the **set** of the natural numbers in Martin-Löf's intuitionistic type theory [Martin-Löf, 1984b; Nordström et al., 1990]. Expressed in the style of Gentzen's natural deduction system, we are justified by our intuition to have the **formation rule**

$$\frac{}{\mathsf{Nat} \;:\; \mathsf{Set}}$$

which says we can conclude (below the line) that Nat is a set from no assumptions (above the line), and the two **introduction rules**

$$\frac{}{\mathsf{zero} \;:\; \mathsf{Nat}} \qquad \frac{n \;:\; \mathsf{Nat}}{\mathsf{suc}\; n \;:\; \mathsf{Nat}}$$

specifying the **canonical elements** of Nat, i.e., those elements that are immediately recognisable as belonging to Nat, namely zero and $\mathsf{suc}\; n$ whenever n is an element of Nat. There are natural numbers not in canonical form, like 10^{10}, but instead encoding an effective method for computing a canonical element. We accept them as **non-canonical elements** of Nat, as long as they compute to a canonical form so we can see that they are indeed natural numbers. Thus, to form a set, we should be able to recognise its elements, either directly or indirectly, as bearing a certain form and thus belonging to the set, so the elements of the set are intuitively clear to us as a certain type of mental constructions.

What is more characteristic of intuitionism is that the intuitionistic interpretation of propositions, and in particular the logical constants, follows the same line of thought as the formation of the set of natural numbers. A proposition is an expression of its truth condition, and since intuitionistic truth follows from proofs, a proposition is clearly specified if and only if what constitutes a proof of it is determined [Martin-Löf, 1987]. What is a proof of a proposition, then? It is a piece of mental construction such that, upon inspection, the truth of the proposition is immediately recognised. For a simple example, in type theory we can formulate the formation rule for disjunctions

$$\frac{A : \mathsf{Set} \qquad B : \mathsf{Set}}{A \vee B : \mathsf{Set}}$$

and the introduction rules

$$\frac{a : A}{\mathsf{inl}\, a : A \vee B} \qquad \frac{b : B}{\mathsf{inr}\, b : A \vee B}$$

saying that a proof (element) of $A \vee B$ is either a proof (element) of A tagged with inl or a proof (element) of B tagged with inr. This is the intuitive (canonical) way we admit as proving a disjunction, and any other (non-canonical) way of proving a disjunction must effectively yield a proof in either of the two forms. The relationship between a proposition and its proofs is thus exactly the same as the one between a set and its elements, namely the proofs must be effectively recognisable as proving the proposition. Hence type theory identifies propositions with sets and proofs with elements, which reflects the observation that proofs are nothing but a certain kind of mental construction.

One notices that the notion of effective method, or computation, was presumed when the notion of set was introduced, and at some point we need to concretely specify an effective method. Since the description of every set includes an effective way to construct its canonical elements, it is possible to express an effective method that mimics the construction of an element by saying that the computation has the same shape as how the element is constructed. Again let us look at the natural numbers. Suppose we have a **family of sets** $P : \mathsf{Nat} \to \mathsf{Set}$ indexed by elements of Nat. The elements of Nat are used as names for these sets, and $P\ n$ denotes the set referred to by the name $n : \mathsf{Nat}$. If we have an element z of P zero and a method s that, for any $n : \mathsf{Nat}$, transforms an element of $P\ n$ to an element of $P\ (\mathsf{suc}\ n)$, then we can compute an element of $P\ n$ for any given n by essentially the same counting process with which we construct n, but the counting now starts from z instead of zero and

higher-order notation

proceeds with s instead of suc. For instance, if a proof of $P\ 2$ is required, we can simply apply s to z twice, just like we apply suc to zero twice to form 2, so the computation was guided by the shape of 2. This explanation justifies the following **elimination rule**

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n) \qquad n : \mathsf{Nat}}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ n : P\ n}$$

The symbol $\mathsf{Nat\text{-}elim}$ symbolises the method described above, which, given P, z, and s, transforms every natural number n into something else of type $P\ n$. If we recall that propositions are identified with sets in type theory, then families of sets like P correspond to predicates, and we see that $\mathsf{Nat\text{-}elim}$ implements exactly the induction principle for natural numbers, as it delivers a proof of $P\ n$ for every $n : \mathsf{Nat}$ if the base case and the inductive case can be proved. The actual computation performed by $\mathsf{Nat\text{-}elim}$ is stated as two **computation rules** in the form of equality judgements:

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n)}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ \mathsf{zero} = z \in P\ \mathsf{zero}}$$

and

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n) \qquad n : \mathsf{Nat}}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ (\mathsf{suc}\ n) = s\ n\ (\mathsf{Nat\text{-}elim}\ P\ z\ s\ n) \in P\ (\mathsf{suc}\ n)}$$

In general, judgemental equality is extended to a congruence relation, so substitutions of equal subterms can be done freely, in particular replacing computations with their results. Note that we only specify how $\mathsf{Nat\text{-}elim}$ computes when it is applied to zero or $\mathsf{suc}\ n$, i.e., the canonical elements, because we have assumed that we can compute a canonical form for each non-canonical element.

We have specified the set of natural numbers by stating its

- formation rule,
- introduction rules,
- elimination rule, and
- computation rules.

The central roles in type theory are played by various sets specified in this manner, corresponding to the various mental constructions we play with in mathematics. Martin-Löf himself noted: "If programming is understood [...] as the design of the methods of computation [...], then it no longer seems possible to distinguish the discipline of programming from constructive mathematics" [Martin-Löf, 1984a]. Indeed, sets are easily comparable with algebraic datatypes, which also play a central role in functional programming — the formation rule names the datatype, the introduction rules list its constructors, and the elimination rule and computation rules define the fold function. One can give concrete, computational explanations for all the entities appearing in type theory, as we have done for the natural numbers, so type theory serves as a suitable foundation for intuitionistic mathematics, which equates mathematical activities with mental constructions.

For the programming side, type theory reveals a new possibility by incorporating logical entities, i.e., propositions and proofs, into the computational world. Traditional theories employ a standalone logic language which is then used to talk about some postulated objects. For example, Peano arithmetic is set up by postulating axioms about natural numbers in the language of first-order logic. Inside the postulated system of natural numbers, there is no knowledge of logic formulas or proofs except via exotic encodings — logic is at a higher level than the objects they are used to talk about. Programming systems based on such principle then need to have a meta-level logic language to reason about properties of programs. In **dependently typed** programming languages based on type theory, however, proving a proposition P is the same as regarding P as a type and then writing programs of that type. The two traditional levels are coherently integrated into one, so programs can be naturally constructed along with their correctness proofs. For example, the proposition $\forall\,(a : A).\ \exists\,(b : B).\ R\ a\ b$ is interpreted as the type of a function taking $a : A$ to a pair consisting of $b : B$ and a proof of the proposition $R\ a\ b$. Once a program typechecks against the type, we are sure that the input and (the first component of) the output of the program are related by R, and the correctness proof is embedded in the program, as opposed to being presented at a meta-level.

Dependently typed programming is thus regarded as a promising way to establish program correctness by construction, but the original formulation of type theory does not offer a convenient language for practical programming. Below we discuss several important revisions which have shown up on the route from type theory to a practical programming language: the adaptation of pattern matching for dependent types, the use of inductive families to manipulate data and their invariants simultaneously, and the quest for a more liberal equality.

2.2 Elimination vs pattern matching

The formation rule and the introduction rules for a set directly translate into an algebraic datatype declaration in functional languages. For example, the type of natural numbers is translated into Agda [Norell, 2009] as

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

Having a datatype, naturally we wish to write programs on that datatype. In functional programming, the pattern matching syntax is widely used for defining programs. It is key to the clarity of functional programs because it not only allows a function to be intuitively defined by several equations but also clearly conveys the strategy of splitting a problem into subproblems by case analysis. On the other hand, computations in type theory are specified using eliminators. Besides keeping the basic theory simple, one reason is that programs in type theory are demanded to be total, for a program must terminate if it is intended as a proof, and using eliminators enforces totality. Pattern matching and elimination are basically equivalent in expressive power, as eliminators can be easily defined by dependent pattern matching, and conversely dependent pattern matching can be reduced to elimination if **uniqueness of identity proofs** — also known as the **K axiom** [Streicher, 1993] — is assumed [Goguen et al., 2006]. Nevertheless, the use of pattern matching together with an interactive development environment is more informative and helpful in dependently typed languages than in simply typed ones, because splitting a problem into subproblems by case analysis in dependently typed programming often leads to nontrivial refinement of the goal type and even the context.

To illustrate, let us look at an example of interactive development in Agda, whose design was inspired by McBride and McKinna [2004]. Consider the following inductively defined less-than-or-equal-to binary relation on natural numbers.

$$
\begin{array}{l}
\textbf{data}\ \_\leqslant\_\ (m : \mathsf{Nat}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{refl} : m \leqslant m \\
\quad \mathit{step} : (n : \mathsf{Nat}) \to m \leqslant n \to m \leqslant \mathsf{suc}\ n
\end{array}
$$

Suppose we are asked to prove that $\_\leqslant\_$ is transitive, i.e., the term

$$\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z$$

can be constructed. We define *trans* interactively by first putting pattern variables for the arguments on the left of its defining equation and leaving an "interaction point" on the right. Agda then tells us a term of type $x \leqslant z$ is expected.

$$
\begin{array}{l}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\mathit{trans}\ x\ y\ z\ p\ q\ =\ \{\, x \leqslant z \,\}_0
\end{array}
$$

We instruct Agda to perform case analysis on q, and there are two cases: refl and $\mathit{step}\ w\ r$ where r has type $y \leqslant w$. The original Goal 0 is split into two sub-goals, and unification is triggered for each sub-goal.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z} \\
\mathit{trans}\ x\ .z\ z & p\ \mathsf{refl} & = & \{\, x \leqslant z \ \|\ p : x \leqslant z \,\}_1 \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w) & p\ (\mathit{step}\ w\ r) & = & \{\, x \leqslant \mathsf{suc}\ w \,\}_2
\end{array}
$$

In Goal 1, the type of refl demands that y be unified with z, and hence the pattern variable y is replaced with a "dot pattern" $.z$ indicating that the value of y is determined by unification to be z. Therefore, on enquiry, Agda tells us that the type of p in the context is now $x \leqslant z$ (which was originally $x \leqslant y$). Similarly for Goal 2, z is unified with $\mathsf{suc}\ w$ and the goal type is rewritten accordingly. We see that the case analysis has led to two subproblems with different goal types and contexts, where Goal 1 is easily solvable as there is a term in the context with the right type, namely p.

$$\begin{array}{lcl}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\mathit{trans}\ x\ .z\ z \quad\quad p\ \mathsf{refl} & = & p \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w)\ p\ (\mathit{step}\ w\ r) & = & \{\, x \leqslant \mathsf{suc}\ w \,\}_2
\end{array}$$

The second goal type $x \leqslant \mathsf{suc}\ w$ looks like the conclusion of $\mathit{step}\ w : x \leqslant w \to x \leqslant \mathsf{suc}\ w$, so we use this term to reduce Goal 2 to Goal 3, which now requires a term of type $x \leqslant w$.

$$\begin{array}{lcl}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\mathit{trans}\ x\ .z\ z \quad\quad p\ \mathsf{refl} & = & p \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w)\ p\ (\mathit{step}\ w\ r) & = & \mathit{step}\ w\ \{\, x \leqslant w \,\}_3
\end{array}$$

Now we see that the induction hypothesis term $\mathit{trans}\ x\ y\ w\ p\ r : x \leqslant w$ (note that r is a sub-term of $\mathit{step}\ w\ r$) has the right type. Filling the term into Goal 3 completes the program.

$$\begin{array}{lcl}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\mathit{trans}\ x\ .z\ z \quad\quad p\ \mathsf{refl} & = & p \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w)\ p\ (\mathit{step}\ w\ r) & = & \mathit{step}\ w\ (\mathit{trans}\ x\ y\ w\ p\ r)
\end{array}$$

In contrast, if we stick to the default elimination approach in type theory, we would be given the eliminator

$$\begin{array}{ll}
\leqslant\text{-elim} : & (m : \mathsf{Nat})\ (P : (n : \mathsf{Nat}) \to m \leqslant n \to \mathsf{Set}) \to \\
& ((t : m \leqslant m) \to P\ m\ t) \to \\
& ((n : \mathsf{Nat})\ (t : m \leqslant n) \to P\ n\ t \to P\ (\mathsf{suc}\ n)\ (\mathit{step}\ n\ t)) \to \\
& (n : \mathsf{Nat})\ (t : m \leqslant n) \to P\ n\ t
\end{array}$$

and write

$$\begin{array}{l}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\mathit{trans}\ x\ y\ z\ p\ q \ = \ \leqslant\text{-elim}\ y\ (\lambda\, y'\ _ \mapsto x \leqslant y \to x \leqslant y') \\
\qquad\qquad\qquad\qquad (\lambda\, _\ p' \mapsto p')\ (\lambda\, w\ r\ \mathit{ih}\ p' \mapsto \mathit{step}\ w\ (\mathit{ih}\ p'))\ z\ q\ p
\end{array}$$

We are forced to write the program in continuation passing style, where the two continuations correspond to the two clauses in the pattern matching version and likewise have more specific goal types, and the relevant context, p in this case, must be explicitly passed into the continuations in order to be refined to a more specific type. Comparing the two versions, we see that elimination is inherently harder to write and understand, especially when complicated dependent types are involved. If a function definition requires more than one level of elimination, then the advantage of using pattern matching over using eliminators becomes even more apparent.

It is often the case that we need to perform pattern matching not only on an argument but also on some intermediate computation. In simply typed languages this is usually achieved by case expressions, a special case being if-then-else expressions for booleans. But again, pattern matching on intermediate computation can make refinements to the goal type and the context in dependently typed languages, so case expressions, being more like eliminators, become less desirable. McBride and McKinna [2004] thus proposed *with*-**matching**, which generalises pattern guards and in effect shifts pattern matching on intermediate computation from the right of an equation to the left, sitting along with the arguments. For a plain example:

$$
\begin{array}{l}
\mathit{insert} \;:\; \mathsf{Nat} \to \mathsf{List\ Nat} \to \mathsf{List\ Nat} \\
\mathit{insert}\ y\ [\,] \;=\; y :: [\,] \\
\mathit{insert}\ y\ (x :: xs)\ \mathbf{with}\ y \leqslant_?\ x \\
\mathit{insert}\ y\ (x :: xs)\ \mid\ \mathit{true} \;\;=\; y :: x :: xs \\
\mathit{insert}\ y\ (x :: xs)\ \mid\ \mathit{false} \;=\; x :: \mathit{insert}\ y\ xs
\end{array}
$$

This is essentially no different from a normal case expression, except that using **with** renders the result of $y \leqslant_? x$ as an additional argument in the context, which is then immediately matched with *true* or *false*. In this case, the original context — y, x, and xs — is not affected by the pattern matching, but in more interesting cases it can be. For example, Wadler's views [1987] can be adapted to dependently typed programming in a more accurate manner, which are supported by **with** in Agda. Suppose we wish to implement a snoc-list view for cons-lists. We define the following view type

$$
\begin{array}{l}
\mathbf{data}\ \mathit{SnocView}\ \{A \;:\; \mathsf{Set}\} \;:\; \mathsf{List}\ A \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{nil} \quad : \mathit{SnocView}\ [\,]
\end{array}
$$

$$\mathsf{snoc} : (xs : \mathsf{List}\ A)\ (x : A) \to \mathit{SnocView}\ (xs +\!\!+ (x :: []))$$

intending to say that a list is either empty or has the form $xs +\!\!+ (x :: [])$, which is proved by the following covering function (whose accuracy is not possible in languages with simpler type disciplines):

$$\begin{array}{lll}
\mathit{snocView} : \{A : \mathsf{Set}\} \to (xs : \mathsf{List}\ A) \to \mathit{SnocView}\ xs & & \\
\mathit{snocView}\ [] = \mathsf{nil} & & \\
\mathit{snocView}\ (x :: xs) & \textbf{with}\ \mathit{snocView}\ xs & \\
\mathit{snocView}\ (x :: .[]) & \mid \mathsf{nil} & = \mathsf{snoc}\ []\ x \\
\mathit{snocView}\ (x :: .(ys +\!\!+ (y :: []))) & \mid \mathsf{snoc}\ ys\ y & = \mathsf{snoc}\ (x :: ys)\ y
\end{array}$$

Then, for example, the function *init* which removes the last element (if any) can be implemented simply as

$$\begin{array}{lll}
\mathit{init} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A & & \\
\mathit{init}\ xs & \textbf{with}\ \mathit{snocView}\ xs & \\
\mathit{init}\ .[] & \mid \mathsf{nil} & = [] \\
\mathit{init}\ .(ys +\!\!+ (y :: [])) & \mid \mathsf{snoc}\ ys\ y & = ys
\end{array}$$

We see that, in both *snocView* and *init*, performing pattern matching on the result of *snocView xs* refines *xs* in the context to either $[]$ or $ys +\!\!+ (y :: [])$ in the two cases. The refined context can be shown explicitly for each case because the matching on *snocView xs* is moved to the left, which is the same difference between using pattern matching and using eliminators. Hence **with**-matching is preferred to traditional case expressions for the same reason that pattern matching is preferred to eliminators: The former clearly expresses context/goal refinements in subproblems in an equational style that is easy to follow, especially when supported by an interactive development environment.

McBride and McKinna [2004] described how programs using pattern matching can be translated into eliminator-based programs. They in fact proposed a general mechanism for invoking any programmer-defined eliminator using the pattern matching syntax, so programmers can choose whichever problem-splitting strategy they need and express that with pattern matching. For example, the standard eliminator for Nat says that to solve a programming problem $P\ n$ for any $n : \mathsf{Nat}$, it is sufficient to solve the more specialised subproblems P zero and P (suc n) (assuming an answer of $P\ n$). This is not the only way to cover all natural numbers, of course; for example, we might split the problem into the two subproblems $P\ i$ where $i < k$ and $P\ (j + k)$ where j : Nat, for some fixed k. We should be able to match a natural number against such nonstandard patterns if that is the strategy we use to divide and solve the problem. Problem specifications can be made more precise by using dependent types, but the solutions would have to be equally precise as a result. Reintroducing pattern matching into dependently typed languages is one step towards helping programmers to describe such solutions naturally and clearly.

2.3 Equality

In logic, the **intension** of a concept is its internal, defining content, while the **extension** of the concept is the range of objects it refers to. In mathematics, for example, the intension of the set $S = \{ x \mid x \in Nat \text{ is even} \}$ is the description that the elements are even natural numbers, and the extension of the set is the enumeration $0, 2, 4, 6, 8, \ldots$. Different intensions may nevertheless lead to the same extension, for example $T = \{ x - 1 \mid x \in Nat \text{ is odd} \}$ is intensionally different from S, but they have the same extension. In other words, S and T use different ways to describe the same range of objects. The axiom of extensionality in set theory defines set equality to be the extensional one, so we consider S and T to be the same set because the extension of S and T are the same, even though they have different intensions. In intuitionistic mathematics, however, the default, fundamental equality is intensional. The reason is that objects in intuitionistic mathematics are given to us as mental **constructions**. For example, the construction of S is to find all the even natural numbers, while the construction of T is to find all the odd natural numbers and subtract 1 from each of them. The two constructions, i.e., descriptions, are different. We can still talk about extensional equality if needed, but that requires a separate definition, which can be a complex proposition in general. For sets, the definition would be $\forall x.\ x \in S \Leftrightarrow x \in T$, i.e., a bi-implication, and we can prove that

Luo [1994]

two sets are extensionally equal in intuitionistic mathematics by proving the bi-implication as we do in classical mathematics. The difference is that in classical mathematics we talk exclusively about extensions and deliberately ignore intensions, so for example “set equality” always refers to the extensional one, whereas in intuitionistic mathematics intensions are also given emphasis. In other words, whereas in both intuitionistic and classical mathematics one can talk about extensionality, an intensional layer about syntactic descriptions of objects is present in intuitionistic mathematics, which is transparent in classical mathematics.

The fundamental equality is formulated as **judgemental equality** in type theory. For intuitionistic mathematics it is the intensional, syntactic equality, also known as **definitional equality**, whereas for classical mathematics it is extensional equality. A characteristic feature of judgemental equality is that it is fully substitutive: judgementally equal terms can be freely substituted for one another. So after we prove that two sets are extensionally equal in classical mathematics, we can simply substitute one for the other because they are judgementally equal in the classical, extensional setting. Judgemental equality cannot be expressed as propositions or have proofs inside the theory, though, since it is a meta-theoretical concept, which, for example, is used in type checking in a language implementation and hence is not an entity in the language. To state equality between two objects as a proposition and have proof for that proposition inside the theory, we need **propositional equality**, which can be defined by the following inductive family.

$\textbf{data}\ \_{\equiv}\_\ \{A : \mathsf{Set}\}\ (x : A) : A \to \mathsf{Set}\ \textbf{where}$
$\quad \mathsf{refl} : x \equiv x$

The canonical way to prove an equality proposition $x \equiv y$ is refl, which is permitted when x and y are judgementally equal. In general, however, computation may be required to prove an equality proposition. For example, the following “catamorphic” identity function on natural numbers

$id' : \mathsf{Nat} \to \mathsf{Nat}$
$id'\ \mathsf{zero} \quad = \mathsf{zero}$

$$id' \; (\mathsf{suc}\; n) \;=\; \mathsf{suc}\; (id'\; n)$$

can be shown to be extensionally equal to the polymorphic identity function

$$\begin{aligned} &id \;:\; \{A \;:\; \mathsf{Set}\} \to A \to A \\ &id\; x \;=\; x \end{aligned}$$

by proving the proposition

$$(n \;:\; \mathsf{Nat}) \to id\; n \;\equiv\; id'\; n$$

whose proof is by induction on n and thus requires computation. It might be said that propositional equality is "delayed" judgemental equality in propositional form: The terms $id\; n$ (which is definitionally just n) and $id'\; n$ are not judgementally equal, but they will compute to the same canonical term (and hence become judgementally equal) after substituting a concrete natural number for n, allowing the computation to complete. Streicher [1993, page 19] suggested that we "consider the identity type [$t \;\equiv\; s$] as a proposition stating a relation between the **objects denoted by the terms** t and s, respectively, whereas the judgement $t \;=\; s \;\in\; A$ is a statement of a relation between the **terms** t and s[.]" Indeed, in an intensional setting, if we regard canonical terms to be the semantic objects denoted by terms, then it might be said that two terms are judgementally equal if their normal forms are syntactically identical, while two terms are propositionally equal if they can be proved to compute to the same canonical term after instantiating the context to canonical terms, i.e., they denote the same semantic object. Practically, when used for substitution, a proof of an equality proposition needs to be eliminated by applying the following standard eliminator commonly called J.

$$\begin{aligned} &J \;:\; \{A \;:\; \mathsf{Set}\}\; \{x \;:\; A\}\; (P \;:\; (y \;:\; A) \to x \equiv y \to \mathsf{Set}) \to \\ &\qquad P\; x\; \mathsf{refl} \to \{y \;:\; A\} \to (eq \;:\; x \equiv y) \to P\; y\; eq \end{aligned}$$

A more convenient substitution operator can be defined in terms of J.

$$\begin{aligned} &subst \;:\; \{A \;:\; \mathsf{Set}\}\; (T \;:\; A \to \mathsf{Set}) \to \{x\, y \;:\; A\} \to x \equiv y \to T\; x \to T\; y \\ &subst\; T\; eq\; t \;=\; J\; (\lambda\, z \mapsto \_)\; T\; z)\; t\; eq \end{aligned}$$

It is like type-casting in programming languages and serves as an explicit proof inside the theory that y can be regarded as x. On the other hand, judgemental equality identifies terms at a more fundamental level and allows a term to be directly substituted for any other term identified with it, without need of any justification inside the theory.

The type of refl says that judgementally equal terms are propositionally equal, so judgemental equality is embedded into propositional equality. If we add the converse **equality reflection rule**

$$\frac{x : A \qquad y : A \qquad eq : x \equiv y}{x = y \in A}$$

to the theory, injecting propositional equality back into judgemental equality, then the resulting type theory is called **extensional**. Type theory without the equality reflection rule is called **intensional**, indicating that its judgemental equality is syntactic. Extensional type theory gets the name because merely syntactic comparison no longer suffices to determine whether two terms are judgementally equal; extensional reasoning may be involved. For example, extensionally equal functions become judgementally equal in extensional type theory: Suppose f and g are functions of type $A \to B$ and we have a proof $fgeq : (x : A) \to f\ x \equiv g\ x$. Then

$$\begin{array}{ll} & f \\ = & \{\ \eta\text{-expansion}\ \} \\ & \lambda\, x \mapsto f\ x \\ = & \{\ \text{equality reflection} — f\ x = g\ x \in B \text{ since } fgeq\ x : f\ x \equiv g\ x\ \} \\ & \lambda\, x \mapsto g\ x \\ = & \{\ \eta\text{-contraction}\ \} \\ & g \qquad \in A \to B \end{array}$$

In general, however, f and g may have very different intensions, so adopting the equality reflection rule makes judgemental equality extensional. The intensional layer present in intuitionistic mathematics thus collapses: The fundamental equality is extensional equality as in classical mathematics, so there is no longer a separate notion of intensional equality. Having extensional equality as judgemental equality makes extensional reasoning much easier because no justification is needed for substitution of extensionally equal terms inside the theory. This is the norm in classical mathematics, where extensionality dominates. For example, in category theory, a universal function (i.e., a universal arrow in the category of sets and total functions) is unique **up to extensional equality**, and category theorists substitute functions satisfying the same universal property for one another all the time. For a language implementation, this means that the programmer can do more than syntactic substitutions without need of explicitly specifying type casts and what equality proofs to use. For example, to show that *id* is extensionally equal to *id*′, we would write the following:

$$\begin{aligned} &\mathit{ideq} : (n : \mathsf{Nat}) \to \mathit{id}\ n \equiv \mathit{id}'\ n \\ &\mathit{ideq}\ \mathsf{zero} \quad = \mathsf{refl} \\ &\mathit{ideq}\ (\mathsf{suc}\ n) = \{\ \mathsf{suc}\ n \equiv \mathsf{suc}\ (\mathit{id}'\ n)\ \} \end{aligned}$$

How the proof is completed depends on whether we are working intensionally or extensionally. If we are working intensionally, the hole needs to be filled with the term

$$\mathit{cong}\ \mathsf{suc}\ (\mathit{ideq}\ n) : \mathsf{suc}\ n \equiv \mathsf{suc}\ (\mathit{id}'\ n)$$

where

$$\mathit{cong} : \{A\ B : \mathsf{Set}\} \to (f : A \to B) \to \{x\ y : A\} \to x \equiv y \to f\ x \equiv f\ y$$

That is, we need to indicate explicitly that we are using an inductively computed result $\mathit{ideq}\ n : n \equiv \mathit{id\_Nat}\ n$, which needs to be further modified by $\mathit{cong}\ \mathsf{suc}$ to match the goal type. On the other hand, if we are working extensionally, a simple refl suffices! The typechecker is told by our placement of refl that $\mathsf{suc}\ n$ and $\mathsf{suc}\ (\mathit{id}'\ n)$ are actually judgementally equal, and has to somehow figure out that there is a term that has type $n \equiv \mathit{id}'\ n$, so n and $\mathit{id}'\ n$ are judgementally equal by equality reflection, and thus $\mathsf{suc}\ n$ and $\mathsf{suc}\ (\mathit{id}'\ n)$ are indeed judgementally equal by congruence. This example illustrates that type checking in an extensional setting cannot simply resort to syntactic equality of normal forms but needs to search for arbitrary equality proofs. The typechecker can ask for hints from the programmer, like in Sheard's Ωmega language [Sheard and Linger, 2007], but type checking becomes undecidable in general. Another perspective to look at this problem is that the rewriting system underlying judgemental equality loses confluence, since different normal forms may be equated due to the equality reflection rule. Consequently, checking syntactic equality of normal forms is no longer a sound way to do type checking. Losing confluence also means that the computational meaning is disrupted since term reduction becomes nondeterministic. Therefore, the reasoning power offered by extensional type theory may be tempting, but to preserve good computational behaviour we have to stick to intensional type theory, namely giving up the equality reflection rule and keeping judgemental equality intensional.

heterogeneous equality, Altenkirch et al. [2007], The Univalent Foundations Program [2013]

2.4 Datatypes and universe construction

Central to **datatype-generic programming** is the idea that the definitional structure of datatypes can be coded as first-class entities and thus become ordinary parameters to programs. The same idea is also found in Martin-Löf's Type Theory [Martin-Löf, 1984b], in which a set of codes for datatypes is called a **universe** (à la Tarski), and there is a decoding function translating codes to actual types. Type theory being the foundation of dependently typed languages, universe construction can be done directly in such languages, so datatype-generic programming becomes just ordinary programming in the dependently typed world [Altenkirch and McBride, 2003]. In this section we construct a universe of **index-first datatypes** [Chapman et al., 2010; Dagand and McBride, 2012b], on which a second universe of **ornaments**, to be constructed in Section 3.2, will depend.

present codes along with their interpretation; not induction-recursion [Dybjer, 1998] though

2.4.1 High-level introduction to index-first datatypes

In Agda, an inductive family is declared by listing all possible constructors and their types, all ending with one of the types in that inductive family. This conveys the idea that the index in the type of an inhabitant is synthesised in a **bottom-up** fashion following the construction of the inhabitant. Consider vectors, for example: the cons constructor takes a vector at some index n and constructs a vector at $\mathsf{suc}\ n$ — the final index is computed bottom-up from the index of the sub-vector. This approach can yield redundant representation, though — the cons constructor for vectors has to store the index of the sub-vector, so the representation of a vector would be cluttered with all the intermediate lengths. If we switch to the opposite perspective, determining **top-down** from the targeted index what constructors should be supplied, then the representation can usually be significantly cleaned up — for a vector, if the index of its type is known to be $\mathsf{suc}\ n$ for some n, then we know that its top-level constructor can only be cons and the index of the sub-vector must be n. To reflect this important reversal of logical order, Dagand and McBride [2012b] proposed a new notation for index-first datatype declarations, in which we first list all possible patterns of (the indices of) the types in the inductive family, and then specify for each pattern which constructors it offers. Below we follow Ko and Gibbons's slightly more Agda-like adaptation of the notation [2013].

Index-first declarations of simple datatypes look almost like Haskell data declarations. For example, natural numbers are declared by

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Nat} : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Nat} \ni \mathsf{zero} \\
\qquad\quad\ \mid \mathsf{suc}\ (n : \mathsf{Nat})
\end{array}
$$

We use the keyword **indexfirst** to explicitly mark the declaration as an index-first one. The only possible pattern of the datatype is Nat, which offers two constructors zero and suc, the latter taking a recursive argument named n. We declare lists similarly, this time with a uniform parameter $A : \mathsf{Set}$:

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{List}\ (A : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{List}\ A \ni [\,]
\end{array}
$$

$\mid \ \_::\_ \ (a : A) \ (as : \mathsf{List}\ A)$

The declaration of vectors is more interesting, fully exploiting the power of index-first datatypes:

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Vec}\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Vec}\ A\ \mathsf{zero} \quad\ \ni\ [\,] \\
\quad \mathsf{Vec}\ A\ (\mathsf{suc}\ n)\ \ni\ \_::\_\ (a : A)\ (as : \mathsf{Vec}\ A\ n)
\end{array}
$$

$\mathsf{Vec}\ A$ is a family of types indexed by Nat, and we do pattern matching on the index, splitting the datatype into two cases $\mathsf{Vec}\ A\ \mathsf{zero}$ and $\mathsf{Vec}\ A\ (\mathsf{suc}\ n)$ for some $n : \mathsf{Nat}$. The first case only offers the nil constructor $[\,]$, and the second case only offers the cons constructor $\_::\_$. Because the form of the index restricts constructor choice, the recursive structure of a vector $as : \mathsf{Vec}\ A\ n$ must follow that of n, i.e., the number of cons nodes in as must match the number of successor nodes in n. We can also declare the bottom-up vector datatype in index-first style:

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Vec}'\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Vec}'\ A\ n\ \ni\ \mathsf{nil}\ (neq : n \equiv \mathsf{zero}) \\
\qquad\qquad\quad \mid\ \mathsf{cons}\ (a : A)\ \{m : \mathsf{Nat}\} \\
\qquad\qquad\qquad\quad (as : \mathsf{Vec}\ A\ m)\ (meq : n \equiv \mathsf{suc}\ m)
\end{array}
$$

Besides the field m storing the length of the tail, two more fields neq and meq are inserted, demanding explicit equality proofs about the indices. When a vector of type $\mathsf{Vec}'\ A\ n$ is demanded, we are "free" to choose between nil or cons regardless of the index n; however, because of the equality constraints, we are indirectly forced into a particular choice.

Remark (*detagging*). The transformation from bottom-up vectors to top-down vectors is exactly what Brady et al.'s **detagging** optimisation [2004] does. With index-first datatypes, however, detagged representations are available directly, rather than arising from a compiler optimisation. □

Remark (*bidirectional typechecking*).

TBC

□

2.4.2 Universe construction

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

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

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

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

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

$$\textbf{data}\ \mathsf{RDesc}\ (I : \mathsf{Set}) : \mathsf{Set}_1$$

with a decoding function specifying its semantics:

$$[\![\_]\!] : \{I : \mathsf{Set}\} \rightarrow \mathsf{RDesc}\ I \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}$$

Inhabitants of $\mathsf{RDesc}\ I$ will be called **response descriptions**. A function of type $I \rightarrow \mathsf{RDesc}\ I$, then, can be decoded to an endofunctor on $I \rightarrow \mathsf{Set}$, so the type $I \rightarrow \mathsf{RDesc}\ I$ acts as a universe for index-first datatypes. We hence define

$$
\begin{aligned}
&\mathsf{Desc} : \mathsf{Set} \rightarrow \mathsf{Set}_1\\
&\mathsf{Desc}\ I = I \rightarrow \mathsf{RDesc}\ I
\end{aligned}
$$

with decoding function

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

Inhabitants of type $\mathsf{Desc}\ I$ will be called **datatype descriptions**, or **descriptions** for short. Actual datatypes are manufactured from descriptions by the least fixed point operator:

$$\begin{array}{l}\textbf{data}\ \mu\ \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) : I \to \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{con} : \mathbb{F}\ D\ (\mu\ D) \rightrightarrows \mu\ D\end{array}$$

We now define the datatype of response descriptions — which determines the syntax available for defining base functors — and its decoding function:

$$\begin{array}{l}\textbf{data}\ \mathsf{RDesc}\ (I : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where}\\ \quad \mathsf{v}\ : (is : \mathsf{List}\ I) \to \mathsf{RDesc}\ I\\ \quad \sigma\ : (S : \mathsf{Set})\ (D : S \to \mathsf{RDesc}\ I) \to \mathsf{RDesc}\ I\end{array}$$

$$\begin{array}{l}[\![\_]\!] : \{I : \mathsf{Set}\} \to \mathsf{RDesc}\ I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ [\![\ \mathsf{v}\ is\ ]\!]\ X = \mathbb{P}\ is\ X \quad \text{-- see below}\\ [\![\ \sigma\ S\ D\ ]\!]\ X = \Sigma[\, s : S\,]\ [\![\ D\ s\ ]\!]\ X\end{array}$$

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

$$\begin{array}{l}\mathbb{P} : \{I : \mathsf{Set}\} \to \mathsf{List}\ I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ \mathbb{P}\ []\qquad\ X = \top\\ \mathbb{P}\ (i :: is)\ X = X\ i \times \mathbb{P}\ is\ X\end{array}$$

Thus, in a response, given $X : I \to \mathsf{Set}$, we are allowed to form dependent sums (by σ) and the product of a finite number of types in X (via v, suggesting variable positions in the base functor).

Convention. We will informally refer to the index part of a σ as a **field**. Like Σ, we regard σ as a binder and write $\sigma[\, s : S\,]\ D\ s$ for $\sigma\ S\ (\lambda\, s \mapsto D\ s)$. □

Example (*natural numbers*). The datatype of natural numbers is considered to be an inductive family trivially indexed by $\top$, so the declaration of Nat corresponds to an inhabitant of $\mathsf{Desc}\ \top$.

$$\textbf{data}\ \mathsf{ListTag} : \mathsf{Set}\ \textbf{where}\ \text{'}\mathsf{nil}\ \text{'}\mathsf{cons} : \mathsf{ListTag}$$

$$\begin{array}{l}NatD : \mathsf{Desc}\ \top\\ NatD\ \blacksquare\ = \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{'}\mathsf{nil} \mapsto \mathsf{v}\ []\\ \qquad\qquad\qquad\qquad\quad ;\ \text{'}\mathsf{cons} \mapsto \mathsf{v}\ (\blacksquare :: [])\ \}\end{array}$$

The index request is necessarily $\blacksquare$, and we respond with a field of type $\mathsf{ListTag}$

representing the constructor choices. If the field receives ‘nil, then we are constructing zero, which takes no recursive values, so we write $\mathsf{v}\ []$ to end this branch; if the ListTag field receives ‘cons, then we are constructing a successor, which takes a recursive value at index $\blacksquare$, so we write $\mathsf{v}\ (\blacksquare :: [])$. $\square$

Example (*lists*). The datatype of lists is parametrised by the element type. We represent parametrised descriptions simply as functions producing descriptions, so the declaration of lists corresponds to a function taking element types to descriptions.

$$
\begin{aligned}
&\mathit{ListD} : \mathsf{Set} \to \mathsf{Desc}\ \top \\
&\mathit{ListD}\ A\ \blacksquare \ = \ \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \ \mapsto\ \mathsf{v}\ [] \\
&\qquad\qquad\qquad\qquad\qquad ;\ \text{‘}\mathsf{cons}\ \mapsto\ \sigma[\,\_ : A\,]\ \mathsf{v}\ (\blacksquare :: [])\ \}
\end{aligned}
$$

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

Example (*vectors*). The datatype of vectors is parametrised by the element type and (non-trivially) indexed by Nat, so the declaration of vectors corresponds to

$$
\begin{aligned}
&\mathit{VecD} : \mathsf{Set} \to \mathsf{Desc}\ \mathsf{Nat} \\
&\mathit{VecD}\ A\ \mathsf{zero} \qquad = \ \mathsf{v}\ [] \\
&\mathit{VecD}\ A\ (\mathsf{suc}\ n) \ = \ \sigma[\,\_ : A\,]\ \mathsf{v}\ (n :: [])
\end{aligned}
$$

which is directly comparable to the index-first base functor VecF' at the beginning of Section 2.4.2. $\square$

There is no problem defining functions on the encoded datatypes except that it has to be done with the raw representation. For example, list append is defined by

$$
\begin{aligned}
&\_{+\!\!+}\_ : \mu\ (\mathit{ListD}\ A)\ \blacksquare \to \mu\ (\mathit{ListD}\ A)\ \blacksquare \to \mu\ (\mathit{ListD}\ A)\ \blacksquare \\
&\mathsf{con}\ (\text{‘}\mathsf{nil}\ \ ,\qquad\quad \blacksquare)\ {+\!\!+}\ bs \ = \ bs \\
&\mathsf{con}\ (\text{‘}\mathsf{cons}\ , a\ , as\ , \blacksquare)\ {+\!\!+}\ bs \ = \ \mathsf{con}\ (\text{‘}\mathsf{cons}\ , a\ , as\ {+\!\!+}\ bs\ , \blacksquare)
\end{aligned}
$$

To improve readability, we define the following higher-level terms:

$$
\begin{aligned}
&\mathsf{List} : \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{List}\ A \ = \ \mu\ (\mathit{ListD}\ A)\ \blacksquare
\end{aligned}
$$

mutual

$$
\begin{aligned}
&\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X)\\
&\mathit{fold}\ \{I\}\ \{D\}\ f\ \{i\}\ (\mathsf{con}\ \mathit{ds}) = f\ (\mathit{mapFold}\ D\ (D\ i)\ f\ \mathit{ds})\\
&\mathit{mapFold} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ (D' : \mathsf{RDesc}\ I) \to\\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to [\![\ D'\ ]\!]\ (\mu\ D) \to [\![\ D'\ ]\!]\ X\\
&\mathit{mapFold}\ D\ (\mathsf{v}\ [])\qquad f\ \blacksquare \qquad = \blacksquare\\
&\mathit{mapFold}\ D\ (\mathsf{v}\ (i :: \mathit{is}))\ f\ (d\ , \mathit{ds}) = \mathit{fold}\ f\ d\ , \mathit{mapFold}\ D\ (\mathsf{v}\ \mathit{is})\ f\ \mathit{ds}\\
&\mathit{mapFold}\ D\ (\sigma\ S\ D')\quad f\ (s\ , \mathit{ds}) = s\ , \mathit{mapFold}\ D\ (D'\ s)\ f\ \mathit{ds}
\end{aligned}
$$

Figure 2.1 Definition of the datatype-generic *fold* operator.

$$
\begin{aligned}
&[] : \{A : \mathsf{Set}\} \to \mathsf{List}\ A\\
&[] = \mathsf{con}\ (\text{‘}\mathsf{nil}\ , \blacksquare)\\
&\_::\_ : \{A : \mathsf{Set}\} \to A \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&a :: \mathit{as} = \mathsf{con}\ (\text{‘}\mathsf{cons}\ , a\ , \mathit{as}\ , \blacksquare)
\end{aligned}
$$

List append can then be rewritten in the usual form (assuming that the terms [] and $\_::\_$ can be used in pattern matching):

$$
\begin{aligned}
&\_{+\!\!+}\_ : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&[]\qquad\quad {+\!\!+}\ \mathit{bs} = \mathit{bs}\\
&(a :: \mathit{as})\ {+\!\!+}\ \mathit{bs} = a :: (\mathit{as}\ {+\!\!+}\ \mathit{bs})
\end{aligned}
$$

Later on, when an encoded datatype is defined, we almost always supply a corresponding index-first datatype declaration immediately afterwards, which is thought of as giving definitions of higher-level terms for type and data constructors — the terms List, [], and $\_::\_$ above, for example, can be considered to be defined by the index-first declaration of lists given in Section 2.4.1. Index-first declarations will only be regarded in this thesis as informal hints at how encoded datatypes are presented at a higher level; we do not give a formal treatment of the elaboration process from index-first declarations to corresponding descriptions and definitions of higher-level terms. (One such treatment was given by Dagand and McBride [2012a].)

Direct function definitions by pattern matching work fine for individual datatypes, but when we need to define operations and to state properties for all the datatypes encoded by the universe, it is necessary to have a generic *fold* operator parametrised by descriptions:

$$\mathit{fold} \; : \; \{I \; : \; \mathsf{Set}\} \; \{D \; : \; \mathsf{Desc}\; I\} \; \{X \; : \; I \rightarrow \mathsf{Set}\} \rightarrow (\mathbb{F}\; D\; X \rightrightarrows X) \rightarrow (\mu\; D \rightrightarrows X)$$

There is also a generic *induction* operator, which can be used to prove generic propositions about all encoded datatypes and subsumes *fold*, but *fold* is much easier to use when the full power of *induction* is not required. The implementations of both operators are adapted for our two-level universe from those in McBride's original work [2011]. We look at the implementation of the *fold* operator only, which is shown in Figure 2.1. As McBride, we would have wished to define *fold* by

$$\mathit{fold}\; \{I\}\; \{D\}\; f\; \{i\}\; (\mathsf{con}\; ds) \;=\; f\; (\mathit{mapRD}\; (D\; i)\; (\mathit{fold}\; f)\; ds)$$

where the functorial mapping *mapRD* on response structures is defined by

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{mapRD} \; : \; \{I \; : \; \mathsf{Set}\} \; (D \; : \; \mathsf{RDesc}\; I) \rightarrow} \\
\multicolumn{4}{l}{\qquad\qquad \{X\; Y \; : \; I \rightarrow \mathsf{Set}\} \; (g \; : \; X \rightrightarrows Y) \rightarrow [\![\, D \,]\!]\; X \rightarrow [\![\, D \,]\!]\; Y} \\
\mathit{mapRD}\; (\mathsf{v}\; []) & g\; \blacksquare & = & \blacksquare \\
\mathit{mapRD}\; (\mathsf{v}\; (i :: is)) & g\; (x\, ,\, xs) & = & g\; x\, ,\, \mathit{mapRD}\; (\mathsf{v}\; is)\; g\; xs \\
\mathit{mapRD}\; (\sigma\; S\; D) & g\; (s\, ,\, xs) & = & s\, ,\, \mathit{mapRD}\; (D\; s)\; g\; xs
\end{array}$$

Agda does not see that this definition of *fold* is terminating, however, since the termination checker does not expand the definition of *mapRD* to see that *fold f* is applied to structurally smaller arguments. To make termination obvious, we instead define *fold* mutually recursively with *mapFold*, which is *mapRD* specialised by fixing its argument g to *fold f*.

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

$$
\begin{array}{l}
\mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ a\ ,\\
\mathsf{con}\ (\text{‘}\mathsf{cons}\ ,\ b\ ,\\
\mathsf{con}\ (\text{‘}\mathsf{nil}\quad ,\\
\quad \blacksquare)\ ,\ \blacksquare)\ ,\ \blacksquare)
\end{array}
$$

Ignoring the last line of trailing $\blacksquare$'s, things following con on each line — including constructor tags and list elements — are shaped by the base functor of lists, whereas the con nodes, aligned vertically, are generated by the least fixed point. This two-dimensional metaphor will be referred to in later explanations.

Remark (*first-order vs higher-order representation*)**.** The functorial structures generated by descriptions are strongly reminiscent of **indexed containers** [Altenkirch and Morris, 2009]; this will be explored and exploited in Chapter 6. For now, it is enough to mention that we choose to stick to a first-order datatype manufacturing scheme, i.e., the datatypes we manufacture with descriptions use finite product types rather than dependent function types for branching, but it is easy to switch to a higher-order representation that is even closer to indexed containers (allowing infinite branching) by storing in v a collection of I-indices indexed by an arbitrary set S:

$$\mathsf{v}\ :\ (S\ :\ \mathsf{Set})\ (f\ :\ S \to I) \to \mathsf{RDesc}\ I$$

whose semantics is defined in terms of dependent functions:

$$[\![\ \mathsf{v}\ S\ f\ ]\!]\ X\ =\ (s\ :\ S) \to X\ (f\ s)$$

The reason for choosing to stick to first-order representation is simply to obtain a simpler equality for the manufactured datatypes (Agda's default equality would suffice); the examples of manufactured datatypes in this thesis are all finitely branching and do not require the power of higher-order representation anyway. This choice, however, does complicate some subsequent datatype-generic definitions (e.g., ornaments). It would probably be helpful to think of the parts involving v and $\mathbb{P}$ in these definitions as specialisations of higher-order representations to first-order ones. □

2.5 Internalism vs externalism

The use of "such that" to describe objects that have certain properties is universal in mathematics. If the objects in question have type A, then objects with certain properties form a subset of A, and using "such that" to describe such objects means that the subset is formed by specifying a suitable predicate on A. In type theory, this can be modelled by the **dependent pair** type.

$$\begin{array}{l} \textbf{data}\ \Sigma\ (A\ :\ \mathsf{Set})\ (B\ :\ A \to \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where} \\ \quad \_,\_\ :\ (x\ :\ A) \to B\ x \to \Sigma\ A\ B \end{array}$$

When A is interpreted as a ground set and B as a predicate on A, an element of $\Sigma\ A\ B$ is an element x of A paired with a proof that $B\ x$ holds. For example, lists of elements of type A with a certain length n are specified by

$$\Sigma\ (\mathsf{List}\ A)\ (\lambda\, xs \mapsto \mathit{length}\ xs\ \equiv\ n)$$

where $\mathit{length}\ :\ \{A\ :\ \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{Nat}$ computes the length of a list. This Σ-type can be naturally read as "the lists xs such that the length of xs is n", bearing some similarity to the notation of set comprehension. Besides being deeply rooted in mathematical traditions, in practice this approach offers very good composability: Whenever a new property is needed, the programmer simply defines a new predicate and uses a Σ-type to impose that predicate on an existing datatype. Predicates easily compose by pointwise conjunction, so objects with two or more properties can be conveniently specified. When programs are the objects we reason about, this style naturally suggests a logical distinction between programs and proofs: Programs are written in the first place, and proofs are conducted afterwards with reference to existing programs and do not interfere with their execution. Consequently, proofs may be erased as they are irrelevant to the computational behaviour of programs. This conception underlies many developments in type theory and theorem proving. For example, Luo [1994] consistently argued that proofs should not be identified with programs, one of the reasons being that logic should be regarded as independent from the objects being reasoned about. A subset theory was described by Nordström et al. [1990] to suppress the second component, i.e., the proof part, of Σ-types. The proof assistant Coq [Bertot and Castéran, 2004] is also designed to support this proving-after-programming style, which is also famous for supporting program extraction from proof scripts [Paulin-Mohring, 1989].

On the other hand, proponents of dependently typed programming believe that, instead of regarding dependent types as yet another type system we impose on existing programs, we should rethink about what programs can be written in a dependently typed language. One such reconsideration is the movement of using inductive families directly for representing data with constraints. The classic example is vectors, which are lists indexed by their length.

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

A simple inductive argument shows that a vector of type Vec A n must be of length n. This fact holds for a vector **by construction**, in contrast to the previous approach using Σ-types, where the length statement is made about a plain list already constructed.

use the new version of explanations of the names internalism and externalism

To illustrate why it can be beneficial to switch from externalism to internalism, suppose we wish to extract the head element from a nonempty list.

draw from the papers

Perhaps not surprisingly, internalist reasoning is rarely seen in mathematics. Here are two possible explanations. The first, philosophical one is that the platonist character of classical mathematics, i.e., the presupposition that mathematical objects are independently existing entities, naturally leads to externalism. The mathematical objects exist *a priori*, and then our proofs are written about them. There is thus a clear "phase distinction," which makes it strange to mix proofs with objects. The second, practical explanation (which also works for non-platonist mathematics) is that it is hard to justify the correctness of an internalist program in prose without silently converting the internalist program to an externalist one. For example, we would say "a vector of length n" and go on about how its length relates to the result, etc., which is not so different from saying "a **list** of length n" and so on — we still need to talk and reason about the constraints separately, unlike how we write an internalist program, which only manipulates data. It might be said that the correctness proof of an internalist program is more syntactic in nature, which in general is more suitable for being checked by machines, while mathematical writing aims to describe the intuition behind so human readers can get the high-level ideas. Even if correctness is implied by the syntactic structure, it is still desirable to have an intuitive explanation of why it is so in prose. Therefore, as the same degree of explanation is needed no matter whether constraints are integrated into syntax or not, it is reasonable to just keep the syntax simple, refraining from using internalist types in mathematical writing.

Type theory makes it possible to mix programs with logic, and thus allows us to bind and manipulate data and proofs together, which is quite different from the programming styles we are used to. We do not know how much potential internalism has, but as its possibility remains to be explored, it seems premature to stick to the traditional, externalist approach that strictly separates programming from logic. Another supporting example is that optimisation of dependently typed programs may exploit value dependencies in types and eliminate a substantial portion of code [Brady et al., 2004] — the length of a vector does not need to be actually stored, *vhead* can just assume that the input vector starts with a cons node, etc. Program optimisation thus does not necessarily take the form of program extraction, which is based on the distinction between programs and proofs. As McBride [2004] said, "[t]here is a tendency to see programming as a fixed notion, essentially untyped. In this view, we make sense of and organise programs by assigning types to them, the way a biologist classifies species, and in order to classify more the exotic creatures, like *printf* or the *zipWith* family, one requires more exotic types. This conception fails to engage with the full potential of types to make a positive contribution to program construction". To support this belief, he presented a development of the first-order unification algorithm, which has long been described using general recursion and required separate termination and correctness proofs, as a structurally recursive, dependently typed program which is correct by construction [McBride, 2003]. The moral is that, to truly adopt internalism, we need to reconsider how we design datatypes and write programs, so their correctness are simply manifested in their construction, reducing the need of external justification which can be more awkward to produce.

Chapter 3

Refinements and ornaments

This chapter begins our exploration of the interconnection between internalism and externalism by looking at the analytic direction, i.e., the decomposition of sophisticated types into basic types and predicates on them. (The synthetic direction will have to wait until Chapter 5.) The 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. For example, consider the insertion operation on ordered vectors:

$$\begin{aligned} \mathit{ovinsert} \;:\; & (x \;:\; \mathit{Val}) \to \{b \;:\; \mathit{Val}\}\ \{n \;:\; \mathsf{Nat}\} \to \mathsf{OrdVec}\ b\ n \to \\ & \{b' \;:\; \mathit{Val}\} \to b' \leqslant x \to b' \leqslant b \to \mathsf{OrdVec}\ b'\ (\mathsf{suc}\ n) \end{aligned}$$

As long as OrdVec is formally unrelated to OrdList and Vec, we cannot reuse insertion on OrdList and Vec but can only reimplement *ovinsert* completely. Here one way to relate the three datatypes is to switch to externalism so it becomes apparent that the datatypes have common ingredients. If we change the appearances of OrdVec in the type of *ovinsert* to its externalist counterpart,

$$\begin{aligned} \mathit{ovinsert}' \;:\; & (x \;:\; \mathit{Val}) \to \{b \;:\; \mathit{Val}\}\ \{n \;:\; \mathsf{Nat}\} \to \\ & \quad \Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,]\ \mathsf{Ordered}\ b\ \mathit{xs} \times \mathit{length}\ \mathit{xs} \;\equiv\; n \to \\ & \{b' \;:\; \mathit{Val}\} \to b' \leqslant x \to b' \leqslant b \to \\ & \quad \Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,]\ \mathsf{Ordered}\ b'\ \mathit{xs} \times \mathit{length}\ \mathit{xs} \;\equiv\; \mathsf{suc}\ n \end{aligned}$$

then, given the three functions

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

we can easily combine them to form *ovinsert′*:

$$
\begin{aligned}
\mathit{ovinsert}'\ x\ (\mathit{xs}\ ,\ \mathit{ord\text{-}xs}\ ,\ \mathit{len\text{-}xs})\ b'{\leqslant}x\ b'{\leqslant}b \ = \ &\mathit{insert}\ x\ \mathit{xs}\ , \\
&\mathit{oinsert}'\ x\ \mathit{xs}\ \mathit{ord\text{-}xs}\ b'{\leqslant}x\ b'{\leqslant}b\ , \\
&\mathit{vinsert}'\ x\ \mathit{xs}\ \mathit{len\text{-}xs}
\end{aligned}
$$

All that is left is converting *ovinsert′* to *ovinsert*, which involves switching from the externalist representation back to OrdVec with the help of the family of **conversion isomorphisms**

$$
\mathsf{OrdVec}\ b\ n \ \cong\ \Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,]\ \mathsf{Ordered}\ b\ \mathit{xs} \times \mathit{length}\ \mathit{xs} \equiv n
$$

for all $b : \mathit{Val}$ and $n : \mathsf{Nat}$. Note that the three functions *insert*, *oinsert′*, and *vinsert′* are reusable components that can go into a library of list datatypes — insertion for OrdList and Vec can also be composed from the three functions in the same way as insertion for OrdVec with the help of appropriate conversion isomorphisms.

This chapter develops the abstractions and constructions that facilitate the above externalist composition of internalist operations as follows:

- Conversion isomorphisms are axiomatised as **refinements** (Section 3.1).
- Refinements are coordinated by **upgrades** (Section 3.1.2) to enable switching between internalist and externalist representations in function types.
- A class of refinements are conveniently synthesised by marking differences between datatypes with **ornaments** (Section 3.2), which relate datatype descriptions that are vertically the same but horizontally different.

TBC (should probably sneak in the term “function upgrading” somewhere)

3.1 Refinements

3.1.1 Refinements between individual types

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

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Refinement}\ (X\ Y : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where} \\
\quad \textbf{field} \\
\qquad P : X \to \mathsf{Set} \\
\qquad i\ \ : Y \cong \Sigma\, X\, P \\
\quad \mathit{forget} : Y \to X \\
\quad \mathit{forget} = \mathsf{outl} \circ \mathsf{Iso}.\mathit{to}\ i
\end{array}
$$

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

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

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

as shown in Section 2.5. An ordered list of type $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ b$ can be decomposed into a list $\mathit{as} : \mathsf{List}\ A$ and a proof of type $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ b\ \mathit{as}$ that the list as is ordered and bounded below by b; conversely, a list satisfying

$\mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ b$ can be promoted to an ordered list of type $\mathsf{OrdList}\ A\ \_{\leqslant_A}\_\ b$.
□

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 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 elements than the basic type does. Thus it is more helpful to think that a type is more refined in the sense of being more informative rather than being a subset. □

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

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

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

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

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

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

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

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

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

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

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

Coherence-based definition of refinements

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

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

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

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

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

3.1.2 Upgrades

Refinements are less useful when we move on to function types: the requirement that a conversion isomorphism exists between related function types is too strong, even when we have extensional equality for functions so isomorphisms between function types make more sense. For example, it is not — and should not be — possible to have a refinement from the function type Nat → Nat to the function type List Nat → 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.

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

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

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

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

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

Explain the meaning of this (scoping).

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

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

definition of $*$

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

i.e., we decompose the input list into the underlying natural number and a vector of elements, process the two parts separately with *double* and *duplicate′*, and finally combine the results back to a list. The relationship between the promoted function *duplicate* and the underlying function *double* is characterised by the coherence property [Dagand and McBride, 2012b]

definition of pointwise equality

$$\mathit{double} \circ \mathit{length} \doteq \mathit{length} \circ \mathit{duplicate}$$

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* does.

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

```
record Upgrade (X Y : Set) : Set₁ where
  field
    P : X → Set
    C : X → Y → Set
    u : (x : X) → P x → Y
    c : (x : X) (p : P x) → C x (u x p)
```

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

Upgrades from refinements

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

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

and consequently any refinement gives rise to an upgrade.

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

Composition of upgrades

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

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

Note that there should be a refinement between the source types X and Y, rather than just an upgrade. (As a consequence, we can produce upgrades between curried multi-argument function types but not between higher-order function types.) This is because, as we see in the *double–duplicate* example, we need the ability to decompose the source type Y.

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

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

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

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

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

$$
\begin{array}{l}
\_\rightharpoonup\_ : \{X\ Y\ Z\ W : \mathsf{Set}\} \to \\
\qquad\quad \mathsf{Refinement}\ X\ Y \to \mathsf{Upgrade}\ Z\ W \to \mathsf{Upgrade}\ (X \to Z)\ (Y \to W) \\
r \rightharpoonup s\ =\ \textbf{record} \\
\quad \{\ P\ =\ \lambda f \mapsto (x : X) \to \mathsf{Refinement}.P\ r\ x \to \mathsf{Upgrade}.P\ s\ (f\ x) \\
\quad ;\ C\ =\ \lambda f\ g \mapsto (x : X)\ (y : Y) \to \\
\qquad\qquad\qquad\qquad \mathsf{Refinement}.\mathit{forget}\ r\ y \equiv x \to \mathsf{Upgrade}.C\ s\ (f\ x)\ (g\ y) \\
\quad ;\ u\ =\ \lambda f\ h \mapsto \mathsf{Upgrade}.u\ s\ \_ \circ \mathit{uncurry}\ h \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.i\ r) \\
\quad ;\ c\ =\ \lambda\ \{\ f\ h\ .\_\ y\ \mathsf{refl} \mapsto \textbf{let}\ (x\ ,p)\ =\ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.i\ r)\ y \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ \mathsf{Upgrade}.c\ s\ (f\ x)\ (h\ x\ p)\ \}\ \}
\end{array}
$$

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

$$
\begin{array}{l}
\quad \text{-- the upgraded function type has an extra argument} \\
\mathit{new} \;:\; \{X : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{Y : I \to \mathsf{Set}\} \to \\
\qquad (\forall\, i \to \mathsf{Upgrade}\ X\ (Y\ i)) \to \mathsf{Upgrade}\ X\ ((i : I) \to Y\ i) \\
\mathit{new}\ I\ u \;=\; \textbf{record}\ \{\ P \;=\; \lambda\, x \mapsto \forall\, i \to \mathsf{Upgrade}.P\ (u\ i)\ x \\
\qquad\qquad\qquad\qquad ;\ C \;=\; \lambda\, x\ y \mapsto \forall\, i \to \mathsf{Upgrade}.C\ (u\ i)\ x\ (y\ i) \\
\qquad\qquad\qquad\qquad ;\ u \;=\; \lambda\, x\ p\ i \mapsto \mathsf{Upgrade}.u\ (u\ i)\ x\ (p\ i) \\
\qquad\qquad\qquad\qquad ;\ c \;=\; \lambda\, x\ p\ i \mapsto \mathsf{Upgrade}.c\ (u\ i)\ x\ (p\ i)\ \} \\
\textbf{syntax}\ \mathit{new}\ I\ (\lambda\, i \mapsto u) \;=\; \forall^{+}[\, i : I\,]\ u \\
\quad \text{-- implicit version of } \mathit{new} \\
\mathit{new}' \;:\; \{X : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{Y : I \to \mathsf{Set}\} \to \\
\qquad (\forall\, i \to \mathsf{Upgrade}\ X\ (Y\ i)) \to \mathsf{Upgrade}\ X\ (\{i : I\} \to Y\ i) \\
\mathit{new}'\ I\ u \;=\; \textbf{record}\ \{\ P \;=\; \lambda\, x \mapsto \forall\, \{i\} \to \mathsf{Upgrade}.P\ (u\ i)\ x \\
\qquad\qquad\qquad\qquad ;\ C \;=\; \lambda\, x\ y \mapsto \forall\, \{i\} \to \mathsf{Upgrade}.C\ (u\ i)\ x\ (y\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ u \;=\; \lambda\, x\ p\ \{i\} \mapsto \mathsf{Upgrade}.u\ (u\ i)\ x\ (p\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ c \;=\; \lambda\, x\ p\ \{i\} \mapsto \mathsf{Upgrade}.c\ (u\ i)\ x\ (p\ \{i\})\ \} \\
\textbf{syntax}\ \mathit{new}'\ I\ (\lambda\, i \mapsto u) \;=\; \forall^{+}[[\, i : I\,]]\ u \\
\quad \text{-- the underlying and the upgraded function types have a common argument} \\
\mathit{fixed} \;:\; (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\}\ \{Y : I \to \mathsf{Set}\} \to \\
\qquad (\forall\, 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 \;=\; \textbf{record}\ \{\ P \;=\; \lambda f \mapsto \forall\, i \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ i) \\
\qquad\qquad\qquad\qquad ;\ C \;=\; \lambda f\ g \mapsto \forall\, i \to \mathsf{Upgrade}.C\ (u\ i)\ (f\ i)\ (g\ i) \\
\qquad\qquad\qquad\qquad ;\ u \;=\; \lambda f\ h\ i \mapsto \mathsf{Upgrade}.u\ (u\ i)\ (f\ i)\ (h\ i) \\
\qquad\qquad\qquad\qquad ;\ c \;=\; \lambda f\ h\ i \mapsto \mathsf{Upgrade}.c\ (u\ i)\ (f\ i)\ (h\ i)\ \} \\
\textbf{syntax}\ \mathit{fixed}\ I\ (\lambda\, i \mapsto u) \;=\; \forall[\, i : I\,]\ u \\
\quad \text{-- implicit version of } \mathit{fixed} \\
\mathit{fixed}' \;:\; (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\}\ \{Y : I \to \mathsf{Set}\} \to \\
\qquad (\forall\, 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 \;=\; \textbf{record}\ \{\ P \;=\; \lambda f \mapsto \forall\, \{i\} \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ C \;=\; \lambda f\ g \mapsto \forall\, \{i\} \to \mathsf{Upgrade}.C\ (u\ i)\ (f\ \{i\})\ (g\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ u \;=\; \lambda f\ h\ \{i\} \mapsto \mathsf{Upgrade}.u\ (u\ i)\ (f\ \{i\})\ (h\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ c \;=\; \lambda f\ h\ \{i\} \mapsto \mathsf{Upgrade}.c\ (u\ i)\ (f\ \{i\})\ (h\ \{i\})\ \} \\
\textbf{syntax}\ \mathit{fixed}'\ I\ (\lambda\, i \mapsto u) \;=\; \forall[[\, i : I\,]]\ u
\end{array}
$$

Figure 3.1 More combinators for upgrades.

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

and the upgrade derived from r, we get an upgrade

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

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

Comparison (*functional ornaments*).

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

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

3.1.3 Refinement families

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

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

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

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

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

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

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

relationship with ornaments

3.2 Ornaments

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

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

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

$$\begin{array}{l}
\mathit{length} : \{A : \mathsf{Set}\} \rightarrow \mu\ (\mathit{ListD}\ A) \rightrightarrows \mu\ \mathit{NatD} \\
\mathit{length}\ \{A\} = \mathit{fold}\ (\mathsf{con} \circ \mathit{erase}\ \{A\}\ \{\mu\ \mathit{NatD}\})
\end{array}$$

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

3.2.1 Universe construction

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

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

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

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

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

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

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

By design, this forgetful map necessarily preserves the recursive structure of its input. In terms of the two-dimensional metaphor mentioned towards the end of Section 2.4.2, 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 con ys where $ys : [\![\ E\ j\ ]\!]\ (\mu\ E)$ and assuming that the $(\mu\ E)$–inhabitants at the recursive positions of ys have been inductively transformed into $(\mu\ D \circ e)$–inhabitants, we horizontally modify the resulting structure of type $[\![\ E\ j\ ]\!]\ (\mu\ D \circ e)$ to one of type $[\![\ D\ (e\ j)\ ]\!]\ (\mu\ D)$, which can then be wrapped by con to an inhabitant of type $\mu\ D\ (e\ j)$. The above steps are performed by the **ornamental algebra** induced by O:

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

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

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

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

connection to refinement families

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

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

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

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 mapping 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:

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

There is only one response ornament in *NatD-ListD A* since the datatype of lists is trivially indexed. The constructor tag is preserved (σ $\mathsf{ListTag}$), and, in the cons case, the list element field is marked as additional by Δ. Consequently, the forgetful function

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

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

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

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

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

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

computes the underlying list of a vector. □

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

3.2.2 Ornamental descriptions

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

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

$$
\begin{array}{l}
\mathit{und}\text{-}\mathbb{P} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ (is : \mathsf{List}\ I) \to \mathbb{P}\ is\ (\mathsf{InvImage}\ e) \to \mathsf{List}\ J \\
\mathit{und}\text{-}\mathbb{P}\ []\qquad\quad \blacksquare \qquad\ = []\\
\mathit{und}\text{-}\mathbb{P}\ (i :: is)\ (j , js) = \mathit{und}\ j :: \mathit{und}\text{-}\mathbb{P}\ is\ js
\end{array}
$$

$$
\begin{array}{l}
\mathit{toRDesc} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{RDesc}\ I\} \to \mathsf{ROrnDesc}\ J\ e\ D \to \mathsf{RDesc}\ J \\
\mathit{toRDesc}\ (\mathsf{v}\ \{is\}\ js) = \mathsf{v}\ (\mathit{und}\text{-}\mathbb{P}\ is\ js) \\
\mathit{toRDesc}\ (\sigma\ S\ OD) = \sigma[s : S]\ \mathit{toRDesc}\ (OD\ s) \\
\mathit{toRDesc}\ (\Delta\ T\ OD) = \sigma[t : T]\ \mathit{toRDesc}\ (OD\ t) \\
\mathit{toRDesc}\ (\nabla\ s\ OD) = \mathit{toRDesc}\ OD
\end{array}
$$

$$
\begin{array}{l}
\mathit{toEq}\text{-}\mathbb{P} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ (is : \mathsf{List}\ I)\ (js : \mathbb{P}\ is\ (\mathsf{InvImage}\ e)) \to \mathbb{E}\ e\ (\mathit{und}\text{-}\mathbb{P}\ is\ js)\ is \\
\mathit{toEq}\text{-}\mathbb{P}\ []\qquad\quad \blacksquare \qquad\ = [] \\
\mathit{toEq}\text{-}\mathbb{P}\ (i :: is)\ (j , js) = \mathit{toEq}\ j :: \mathit{toEq}\text{-}\mathbb{P}\ is\ js
\end{array}
$$

$$
\begin{array}{l}
\mathit{toROrn} : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{RDesc}\ I\} \to \\
\qquad\qquad (OD : \mathsf{ROrnDesc}\ J\ e\ D) \to \mathsf{ROrn}\ e\ D\ (\mathit{toRDesc}\ OD) \\
\mathit{toROrn}\ (\mathsf{v}\ js) \qquad = \mathsf{v}\ (\mathit{toEq}\text{-}\mathbb{P}\ \_\ js) \\
\mathit{toROrn}\ (\sigma\ S\ OD) = \sigma[s : S]\ \mathit{toROrn}\ (OD\ s) \\
\mathit{toROrn}\ (\Delta\ T\ OD) = \Delta[t : T]\ \mathit{toROrn}\ (OD\ t) \\
\mathit{toROrn}\ (\nabla\ s\ OD) = \nabla[s]\ (\mathit{toROrn}\ OD)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{OrnDesc} : \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I)\ (D : \mathsf{Desc}\ I) \to \mathsf{Set}_1 \\
\mathsf{OrnDesc}\ J\ e\ D = \{i : I\}\ (j : e^{-1}\ i) \to \mathsf{ROrnDesc}\ J\ e\ (D\ i)
\end{array}
$$

$$
\begin{array}{l}
\lfloor\_\rfloor : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{Desc}\ I\} \to \mathsf{OrnDesc}\ J\ e\ D \to \mathsf{Desc}\ J \\
\lfloor OD \rfloor\ j = \mathit{toRDesc}\ (OD\ (\mathsf{ok}\ j))
\end{array}
$$

$$
\begin{array}{l}
\lceil\_\rceil : \{I\ J : \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{Desc}\ I\}\ (OD : \mathsf{OrnDesc}\ J\ e\ D) \to \mathsf{Orn}\ e\ D\ \lfloor OD \rfloor \\
\lceil OD \rceil\ (\mathsf{ok}\ j) = \mathit{toROrn}\ (OD\ (\mathsf{ok}\ j))
\end{array}
$$

Figure 3.2 Definitions for ornamental descriptions.

would be more efficient if we could use the existing description as a template and just write a "relative description" specifying how to "patch" the template, and afterwards from this "relative description" extract a new description and an ornament from the template to the new description.

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

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

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

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

and the ornament is

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

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

$$\begin{array}{l}
\mathit{OrdListOD}\; A\; _\!\leqslant_A\!_ \;:\; \mathsf{OrnDesc}\; A\; !\; (\mathit{ListD}\; A) \\
\mathit{OrdListOD}\; A\; _\!\leqslant_A\!_\; (\mathsf{ok}\; b) \;=\; \\
\quad \sigma\; \mathsf{ListTag}\; \lambda\; \{\; \text{`nil} \quad \mapsto\; \mathsf{v}\; \blacksquare \\
\qquad\qquad\qquad ;\; \text{`cons} \;\mapsto\; \sigma[\, a : A\,]\;\; \Delta[\, \mathit{leq} : b \leqslant_A a\,]\;\; \mathsf{v}\; (a\; ,\; \blacksquare)\; \}
\end{array}$$

$$\begin{array}{l}
\textbf{indexfirst data}\; \mathsf{OrdList}\; A\; _\!\leqslant_A\!_ \;:\; A \to \mathsf{Set}\; \textbf{where} \\
\quad \mathsf{OrdList}\; A\; _\!\leqslant_A\!_\; b \;\ni\; \mathsf{nil} \\
\qquad\qquad\qquad\qquad\quad \mid\; \mathsf{cons}\; (a \;:\; A)\; (\mathit{leq} \;:\; b \leqslant_A a)\; (\mathit{as} \;:\; \mathsf{OrdList}\; A\; _\!\leqslant_A\!_\; a)
\end{array}$$

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

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

and

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

is an ornament from lists to ordered lists. □

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

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

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

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

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 the pattern. In general, we can define a parametrised ornamental description

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

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

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

$$\mathit{erode}\ (\sigma\ S\ D)\ (s\ ,\ \mathit{js})\ =\ \nabla[\,s\,]\ \mathit{erode}\ (D\ s)\ \mathit{js}$$

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

where

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

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

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

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

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

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

and the "swapping" ornament

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

whose forgetful function swaps the two fields.

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

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

Tagged descriptions are decoded to ordinary descriptions by

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

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

coproduct-related definitions

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

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

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

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

3.2.3 Parallel composition of ornaments

intro — analysis for composability

The generic scenario is illustrated below: Chapter 4

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

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

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

Intuitively, since both O and P encode modifications to the same base description D, we can commit all modifications encoded by O and P to D to get a new description $\lfloor O \otimes P \rfloor$, and encode all these modifications in one ornament $\lceil O \otimes P \rceil$. (This merging of two sets of modifications is best characterised by a category-theoretic pullback, which we defer until Chapter 4.) 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**

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

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

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

and

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

are extensionally equal to *forget* $\lceil O \otimes P \rceil$.

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

$$\begin{array}{l}
\mathsf{OrdVec}\ A\ _{\leqslant_A}_ \ :\ A \to \mathsf{Nat} \to \mathsf{Set} \\
\mathsf{OrdVec}\ A\ _{\leqslant_A}_\ b\ n\ = \\
\quad \mu\ \lfloor \lceil \textit{OrdListOD}\ A\ _{\leqslant_A}_ \rceil \otimes \textit{ListD-VecD}\ A \rfloor\ (\mathsf{ok}\ (\mathsf{ok}\ b\ ,\ \mathsf{ok}\ n))
\end{array}$$

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{OrdVec}\ A\ _{\leqslant_A}_ \ :\ A \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{OrdVec}\ A\ _{\leqslant_A}_\ b\ \mathsf{zero} \quad\quad \ni\ \mathsf{nil} \\
\quad \mathsf{OrdVec}\ A\ _{\leqslant_A}_\ b\ (\mathsf{suc}\ n)\ \ni\ \mathsf{cons}\ (a\ :\ A)\ (\textit{leq}\ :\ b \leqslant_A a) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\textit{as}\ :\ \mathsf{OrdVec}\ A\ _{\leqslant_A}_\ a\ n)
\end{array}$$

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

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

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

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

- 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

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

mutual

$$
\begin{array}{l}
\quad \mathit{pcROD} \,:\, \{I\;J\;K \,:\, \mathsf{Set}\}\;\{e \,:\, J \to I\}\;\{f \,:\, K \to I\} \\
\qquad\qquad \{D \,:\, \mathsf{RDesc}\;I\}\;\{E \,:\, \mathsf{RDesc}\;J\}\;\{F \,:\, \mathsf{RDesc}\;K\} \to \\
\qquad\qquad \mathsf{ROrn}\;e\;D\;E \to \mathsf{ROrn}\;f\;D\;F \to \mathsf{ROrnDesc}\;(e \bowtie f)\;\mathit{pull}\;D
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{pcROD}\;(\mathsf{v}\;\mathit{eeqs}) & (\mathsf{v}\;\mathit{feqs}) & = & \mathsf{v}\;(\mathit{pc}\text{-}\mathbb{E}\;\mathit{eeqs}\;\mathit{feqs}) & \\
\mathit{pcROD}\;(\mathsf{v}\;\mathit{eeqs}) & (\Delta\;T\;P) & = & \Delta[\,t : T\,]\;\mathit{pcROD}\;(\mathsf{v}\;\mathit{eeqs}) & (P\;t) \\
\mathit{pcROD}\;(\sigma\;S\;O) & (\sigma\;.S\;P) & = & \sigma[\,s : S\,]\;\mathit{pcROD}\;(O\;s) & (P\;s) \\
\mathit{pcROD}\;(\sigma\;f\;O) & (\Delta\;T\;P) & = & \Delta[\,t : T\,]\;\mathit{pcROD}\;(\sigma\;f\;O) & (P\;t) \\
\mathit{pcROD}\;(\sigma\;S\;O) & (\nabla\;s\;P) & = & \nabla[\,s\,]\quad\;\; \mathit{pcROD}\;(O\;s) & P \\
\mathit{pcROD}\;(\Delta\;T\;O)\;P & & = & \Delta[\,t : T\,]\;\mathit{pcROD}\;(O\;t) & P \\
\mathit{pcROD}\;(\nabla\;s\;O) & (\sigma\;S\;P) & = & \nabla[\,s\,]\quad\;\; \mathit{pcROD}\;O & (P\;s) \\
\mathit{pcROD}\;(\nabla\;s\;O) & (\Delta\;T\;P) & = & \Delta[\,t : T\,]\;\mathit{pcROD}\;(\nabla\;s\;O) & (P\;t) \\
\mathit{pcROD}\;(\nabla\;s\;O) & (\nabla\;s'\;P) & = & \Delta\;(s \equiv s')\;(\mathit{pcROD\text{-}double}\nabla\;O\;P) &
\end{array}
$$

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

$$
\begin{array}{l}
\_\otimes\_ \,:\, \{I\;J\;K \,:\, \mathsf{Set}\}\;\{e \,:\, J \to I\}\;\{f \,:\, K \to I\} \\
\qquad \{D \,:\, \mathsf{Desc}\;I\}\;\{E \,:\, \mathsf{Desc}\;J\}\;\{F \,:\, \mathsf{Desc}\;K\} \to \\
\qquad \mathsf{Orn}\;e\;D\;E \to \mathsf{Orn}\;f\;D\;F \to \mathsf{OrnDesc}\;(e \bowtie f)\;\mathit{pull}\;D \\
(O \otimes P)\;(\mathsf{ok}\;(j\;,k)) \;=\; \mathit{pcROD}\;(O\;j)\;(P\;k)
\end{array}
$$

Figure 3.3 Definitions for parallel composition of ornaments.

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

- 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

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

- 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,

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

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

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

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

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

  and

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

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

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

  where the added equality field completes the construction of a set-theoretic pullback of g and h. Here indeed we need a pullback: When we have an actual value for the field σ 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. Chapter 4

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

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

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

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

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

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

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

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

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

is indeed extensionally equal to *forget* $\lceil O \otimes P \rceil$, since *forget* (*diffOrn-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.

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

mutual

$$
\begin{array}{l}
\quad \mathit{diffROrn}\text{-}\mathit{l} \;: \\
\qquad \{I\;J\;K \;:\; \mathsf{Set}\}\;\{e \;:\; J \to I\}\;\{f \;:\; K \to I\} \to \\
\qquad \{D \;:\; \mathsf{RDesc}\;I\}\;\{E \;:\; \mathsf{RDesc}\;J\}\;\{F \;:\; \mathsf{RDesc}\;K\} \to \\
\qquad (O \;:\; \mathsf{ROrn}\;e\;D\;E)\;(P \;:\; \mathsf{ROrn}\;f\;D\;F) \to \mathsf{ROrn}\;\mathsf{outl}\;E\;(\mathit{toRDesc}\;(\mathit{pcROD}\;O\;P))
\end{array}
$$

$$
\begin{array}{llllll}
\mathit{diffROrn}\text{-}\mathit{l}\;(\mathsf{v}\;eeqs) & (\mathsf{v}\;feqs) & = & \multicolumn{3}{l}{\mathsf{v}\;(\mathit{diff}\text{-}\mathbb{E}\text{-}\mathit{l}\;eeqs\;feqs)} \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\mathsf{v}\;eeqs) & (\Delta\;T\;P) & = & \Delta[\,t : T\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(\mathsf{v}\;eeqs) & (P\;t) \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\sigma\;S\;O) & (\sigma\;.S\;P) & = & \sigma[\,s : S\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(O\;s) & (P\;s) \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\sigma\;S\;O) & (\Delta\;T\;P) & = & \Delta[\,t : T\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(\sigma\;S\;O) & (P\;t) \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\sigma\;S\;O) & (\nabla\;s\;P) & = & \nabla[\,s\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(O\;s) & P \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\Delta\;T\;O)\;P & & = & \sigma[\,t : T\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(O\;t) & P \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\nabla\;s\;O) & (\sigma\;S\;P) & = & & \mathit{diffROrn}\text{-}\mathit{l}\;O & (P\;s) \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\nabla\;s\;O) & (\Delta\;T\;P) & = & \Delta[\,t : T\,] & \mathit{diffROrn}\text{-}\mathit{l}\;(\nabla\;s\;O) & (P\;t) \\
\mathit{diffROrn}\text{-}\mathit{l}\;(\nabla\;s\;O) & (\nabla\;s'\;P) & = & \multicolumn{3}{l}{\Delta\;(s \equiv s')\;(\mathit{diffROrn}\text{-}\mathit{l}\text{-}\mathit{double}\nabla\;O\;P)}
\end{array}
$$

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

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

Figure 3.4 Definitions for left difference ornament.

3.3 Refinement semantics of ornaments

Every ornament O : $\mathsf{Orn}\ e\ D\ E$ induces a refinement family from $\mu\ D$ to $\mu\ E$. That is, we can construct a function

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

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

intro

3.3.1 Optimised predicates

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

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

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

Optimised in what sense?

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

to a list, an optimised promotion proof would look like

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where lighter box indicates contributions from *ListD-VecD A* and darker box from ⌈ *singletonOD* (*ListD A*) ⌉. 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 λ ().

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

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{indexfirst data}\ \mathsf{Length}\ A : \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where}} \\
\quad \mathsf{Length}\ A\ \mathsf{zero} & [\,] & \ni & \mathsf{nil} \\
\quad \mathsf{Length}\ A\ \mathsf{zero} & (a :: as) & \not\ni & \\
\quad \mathsf{Length}\ A\ (\mathsf{suc}\ n) & [\,] & \not\ni & \\
\quad \mathsf{Length}\ A\ (\mathsf{suc}\ n) & (a :: as) & \ni & \mathsf{cons}\ (l : \mathsf{Length}\ A\ n\ as)
\end{array}
$$

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

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

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

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

$\mathit{ornConvIso}$:

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

The construction of *ornConvIso* will be deferred until Chapter 4.

3.3.2 Predicate swapping for parallel composition

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

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

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

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

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_ : A \to \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_\ b\ \mathsf{zero}\quad\ \ [\,] \qquad\quad \ni\ \mathsf{nil} \\
\quad \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_\ b\ \mathsf{zero}\quad\ \ (a :: as)\ \not\ni \\
\quad \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_\ b\ (\mathsf{suc}\ n)\ [\,] \qquad\quad \not\ni \\
\quad \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_\ b\ (\mathsf{suc}\ n)\ (a :: as) \\
\qquad \ni\ \mathsf{cons}\ (\mathit{leq} : b \leqslant_A a)\ (\mathit{ol} : \mathsf{OrderedLength}\ A\ _\!\leqslant_A\!_\ a\ n\ as)
\end{array}$$

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

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

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

Along with the ornamental conversion isomorphisms, the construction of the modularity isomorphisms will be deferred until Chapter 4. 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:

```
record Swap {X Y : Set} (r : Refinement X Y) : Set₁ where
  field
    Q : X → Set
    i : (x : X) → Refinement.P r x ≅ Q x
```

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

```
toRefinement : {X Y : Set} {r : Refinement X Y} → Swap r → Refinement X Y
toRefinement s = record { P = Swap.Q s
                        ; i = { }₀ }
```

where Goal 0 is the new conversion isomorphism

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

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

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

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 $\lceil O \otimes P \rceil$:

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

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

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

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

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

where

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

simply retains the original promotion predicate in *rs*. □

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

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

yields a refinement family from lists to ordered vectors using

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

as the promotion predicate, where

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

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

3.3.3 Resolution of the list insertion example

3.4 Examples

To further demonstrate the use of the ornament–refinement framework, we look at two dependently typed heap data structures adapted from Okasaki's work on purely functional data structures [1999]. The first example about **binomial heaps** shows that Okasaki's idea of **numerical representations** can be elegantly captured by ornaments and the coherence properties computed with upgrades, and the second example about **leftist heaps** demonstrates the power of parallel composition of ornaments by treating heap ordering and leftist balancing properties modularly.

Postulate operations on *Val* like $\_\leqslant_?\_$, $\leqslant$*-refl*, $\leqslant$*-trans*, and $\nleqslant$*-invert* in Chapter 2.

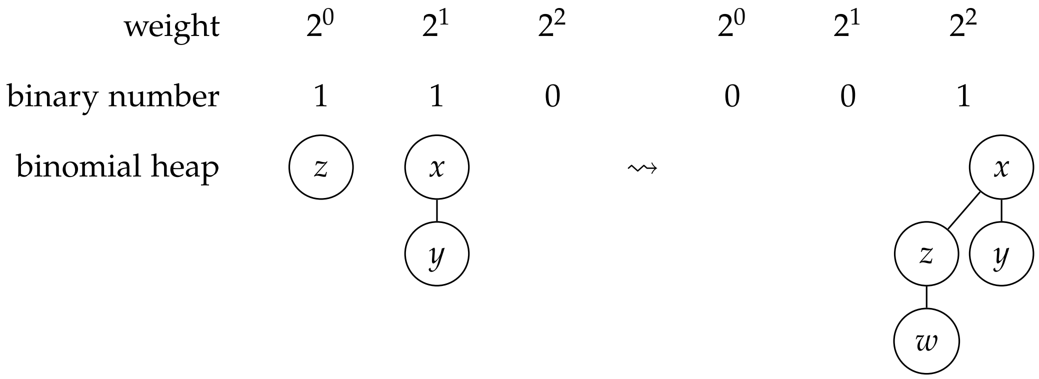

Figure 3.5 *Left:* a binomial heap of size 3 consisting of two binomial trees storing elements x, y, and z. *Right:* the result of inserting an element w into the heap. (Note that the digits of the underlying binary numbers are ordered with the least significant digit first.)

3.4.1 Binomial heaps

We are all familiar with the idea of **positional number systems**, in which we represent numbers as a list of digits. Each position in a list of digits is associated with a weight, and the interpretation of the list is the weighted sum of the digits. (For example, the weights used for binary numbers are powers of 2.) Some container data structures and associated operations strongly resemble positional representations of natural numbers and associated operations. For example, a **binomial heap** (illustrated in Figure 3.5) can be thought of as a binary number in which every 1-digit stores a **binomial tree** — the actual place for storing elements — whose size is exactly the weight of the digit. The number of elements stored in a binomial heap is therefore exactly the value of the underlying binary number. Inserting a new element into a binomial heap is analogous to incrementing a binary number, with carrying corresponding to combining smaller binomial trees into larger ones. Okasaki thus proposed to design container data structures by analogy with positional representations of natural numbers, and called such data structures **numerical representations**. Using an ornament, it is easy to express the relationship between a numerically represented container datatype (e.g., binomial heaps) and its underlying

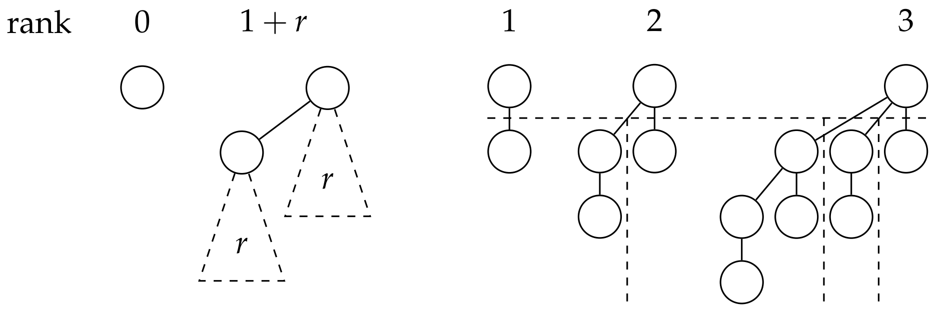

Figure 3.6 *Left:* inductive definition of binomial trees. *Right:* decomposition of binomial trees of ranks 1 to 3.

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. Binomial trees are defined inductively on their **rank**, which is a natural number (see Figure 3.6):

- a binomial tree of rank 0 is a single node storing an element of type *Val*, and
- a binomial tree of rank $1 + r$ consists of two binomial trees of rank r, with one attached under the other's root node.

From this definition we can readily deduce that a binomial tree of rank r has 2^r elements. To actually define binomial trees as a datatype, however, an alternative view is more useful: a binomial tree of rank r is constructed by attaching binomial trees of ranks 0 to $r - 1$ under a root node. (Figure 3.6 shows how binomial trees of ranks 1 to 3 can be decomposed according to this view.) We thus define the datatype BTree : Nat → Set — which is indexed with the rank of binomial trees — as follows: for any rank r : Nat, the type 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:

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

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

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

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

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

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

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

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

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:

$$\begin{array}{l}
\mathbf{data}\ \mathsf{BinTag} : \mathsf{Set}\ \mathbf{where}\ \text{'nil 'zero 'one} : \mathsf{BinTag} \\
\mathit{BinD} : \mathsf{Desc}\ \top \\
\mathit{BinD}\ \blacksquare = \sigma\ \mathsf{BinTag}\ \lambda\ \{\ \text{'nil} \mapsto \mathsf{v}\ [] \\
\qquad\qquad\qquad\qquad\qquad ;\ \text{'zero} \mapsto \mathsf{v}\ (\blacksquare :: []) \\
\qquad\qquad\qquad\qquad\qquad ;\ \text{'one} \mapsto \mathsf{v}\ (\blacksquare :: [])\ \} \\
\mathbf{indexfirst\ data}\ \mathsf{Bin} : \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{Bin} \ni \mathsf{nil} \\
\qquad\quad \mid \mathsf{zero}\ (b : \mathsf{Bin}) \\
\qquad\quad \mid \mathsf{one}\ \ (b : \mathsf{Bin})
\end{array}$$

The intended interpretation of binary numbers is given by

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

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 $1 + r$.

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

indexfirst data BHeap : Nat → Set **where**

$$
\begin{aligned}
\mathsf{BHeap}\ r \ni\ & \mathsf{nil}\\
\mid\ & \mathsf{zero}\ (h : \mathsf{BHeap}\ (\mathsf{suc}\ r))\\
\mid\ & \mathsf{one}\ \ (t : \mathsf{BTree}\ r)\ (h : \mathsf{BHeap}\ (\mathsf{suc}\ r))
\end{aligned}
$$

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

Increment and insertion, in coherence

Increment of binary numbers is defined by

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

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

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

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

Conceptually, *incr* puts a 1-digit into the least significant position of a binary number, triggering a series of 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.

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

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:

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

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

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

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

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

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 $\mathit{incr}\ b$. This says exactly that *insT* manipulates the underlying binary number in the same way as *incr* does.

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

Let BHeap' be the optimised predicate for the ornament from Bin to $\mathsf{BHeap}\ r$:

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

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

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

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

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

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

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

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

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

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

which is automatically in coherence with *incr*:

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

Summary

We define Bin, *incr*, and then BHeap as an ornamentation of Bin, describe in *upg* how the type of *insT* is an upgraded version of the type of *incr*, and implement *insT*′, whose type is supplied by *upg*. We can then derive *insT*, the coherence property of *insT* with respect to *incr*, and its proof, all automatically by *upg*. Compared to Okasaki's implementation, besides rank-indexing, which elegantly transfers the management of rank-related invariants to the type system, the extra work is only the straightforward markings of the differences between Bin and BHeap (in *BHeapOD*) and between the type of *incr* and that of *insT* (in *upg*). The reward is huge in comparison: we get a coherence property that precisely characterises the structural behaviour of insertion with respect to increment, and an enriched function type that guides the implementation of insertion such that the coherence property is satisfied by construction. From straightforward markings to nontrivial types and programs — this clearly demonstrates the power of the ornament–refinement framework.

3.4.2 Leftist heaps

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

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

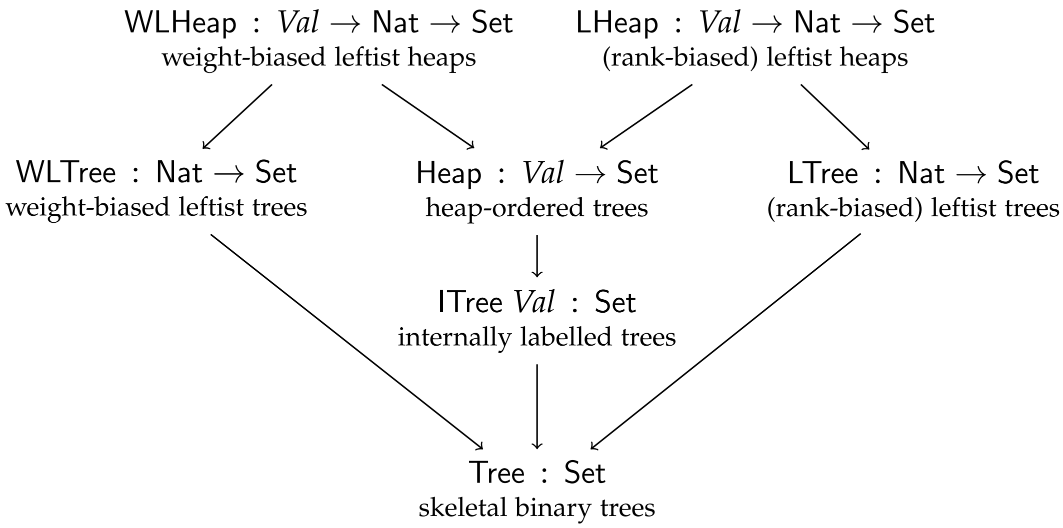

Figure 3.7 Datatypes about leftist heaps and their ornamental relationships.

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

Datatypes leading to leftist heaps

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

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

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

$$\textbf{indexfirst data}\ \mathsf{Tree} : \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Tree} \ni \mathsf{nil}$$
$$\mid \mathsf{node}\ (t : \mathsf{Tree})\ (u : \mathsf{Tree})$$

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

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

$$\textbf{indexfirst data}\ \mathsf{LTree} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Tree}\ \mathsf{zero} \ni \mathsf{nil}$$
$$\mathsf{Tree}\ (\mathsf{suc}\ r) \ni \mathsf{node}\ \{l : \mathsf{Nat}\}\ (r{\leqslant}l : r \leqslant l)\ (t : \mathsf{Tree}\ l)\ (u : \mathsf{Tree}\ r)$$

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

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

$$\textbf{indexfirst data}\ \mathsf{Heap} : \mathit{Val} \to \mathsf{Set}\ \textbf{where}$$
$$\mathsf{Heap}\ b \ni \mathsf{nil}$$
$$\mid \mathsf{node}\ (x : \mathit{Val})\ (b{\leqslant}x : b \leqslant x)\ (t : \mathsf{Heap}\ x)\ (u : \mathsf{Heap}\ x)$$

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

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

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

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

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

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

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

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:

$$
\begin{array}{l}
\mathit{lhrelax} : \{b\ b' : \mathit{Val}\} \to b' \leqslant b \to \{r : \mathsf{Nat}\} \to \mathsf{LHeap}\ b\ r \to \mathsf{LHeap}\ b'\ r \\
\mathit{lhrelax}\ \{b\}\ \{b'\}\ b'{\leqslant}b\ \{r\}\ =\ \mathsf{Upgrade}.\mathit{u}\ \mathit{upg}\ \mathit{id}\ (\lambda\ \_\ \mapsto\ \mathit{relax}\ b'{\leqslant}b * \mathit{id})
\end{array}
$$

where

$$
\begin{aligned}
&\mathit{ref} \;:\; (b'' \;:\; \mathit{Val}) \to \mathsf{Refinement\ Tree}\ (\mathsf{LHeap}\ b''\ r) \\
&\mathit{ref}\ b'' \;=\; \mathit{toRefinement} \\
&\qquad\qquad (\otimes\text{-}\mathit{FSwap}\ \lceil \mathit{HeapOD} \rceil\ \lceil \mathit{LTreeOD} \rceil\ \mathit{id}\text{-}\mathit{FSwap}\ \mathit{id}\text{-}\mathit{FSwap} \\
&\qquad\qquad\quad (\mathsf{ok}\ (\mathsf{ok}\ b''\ , \mathsf{ok}\ r))) \\
&\mathit{upg} \;:\; \mathsf{Upgrade}\ (\mathsf{Tree} \to \mathsf{Tree})\ (\mathsf{LHeap}\ b\ r \to \mathsf{LHeap}\ b'\ r) \\
&\mathit{upg} \;=\; \mathit{ref}\ b \rightharpoonup \mathit{toUpgrade}\ (\mathit{ref}\ b')
\end{aligned}
$$

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 has nothing to do with the leftist property. In fact, it even has nothing to do with heap ordering, but only with the internal labelling. Hence we can define the **internally labelled trees** as an ornamentation of skeletal binary trees:

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

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

on which we can do preorder traversal:

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

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

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

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

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

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

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

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

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

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

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

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

Weight-biased leftist heaps

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

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

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

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

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

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

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

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

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

Do we need a summary here?

3.5 Discussion

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

Chapter 4

Categorical organisation of the ornament–refinement framework

Chapter 3 left some obvious holes in the theory of ornaments. For instance:

- When it comes to composition of ornaments, the following **sequential composition** is probably the first that comes to mind (rather than parallel composition), which is evidence that the ornamental relation is transitive:

  $$\begin{aligned} \_\odot\_ \; : \; & \{I\;J\;K \; : \; \mathsf{Set}\}\; \{e \; : \; J \to I\}\; \{f \; : \; K \to J\} \to \\ & \{D \; : \; \mathsf{Desc}\; I\}\; \{E \; : \; \mathsf{Desc}\; J\}\; \{F \; : \; \mathsf{Desc}\; K\} \to \\ & \mathsf{Orn}\; e\; D\; E \to \mathsf{Orn}\; f\; E\; F \to \mathsf{Orn}\; (e \circ f)\; D\; F \end{aligned}$$

  -- definition in Figure 4.4

  Correspondingly, we expect that

  $$\mathit{forget}\;(O \odot P) \qquad \text{and} \qquad \mathit{forget}\; O \circ \mathit{forget}\; P$$

  are extensionally equal. That is, the sequential compositional structure of ornaments corresponds to the compositional structure of forgetful functions. We wish to state such correspondences in concise terms.

- While parallel composition of ornaments has a sensible definition (Section 3.2.3), it is defined by case analysis at the microscopic level of individual fields. Such a microscopic definition is difficult to comprehend, and so are any subsequent definitions and proofs. It is desirable to have a macroscopic characterisation of parallel composition, so the nature of parallel composition is immediately clear, and subsequent definitions and proofs can be done in a more abstract manner.

- The ornamental conversion isomorphisms (Section 3.3.1) and the modularity isomorphisms (Section 3.3.2) were left unimplemented. Both sets of isomorphisms are about the optimised predicates (Section 3.3.1), which are defined in terms of parallel composition with singleton ornaments (Section 3.2.2). We thus expect that the existence of these isomorphisms can be explained in terms of properties of parallel composition and singleton ornaments.

A lightweight organisation of the ornament–refinement framework in basic category theory [Mac Lane, 1998] can help to fill up all these holes. In more detail:

- Categories and functors are abstractions for compositional structures and structure-preserving maps between them. Facts about translations between ornaments, refinements, and functions can thus be neatly organised under the categorical language (Section 4.1).
- Parallel composition merges two compatible ornaments and does nothing more; in other words, it computes the least informative ornament that contains the information of both ornaments. Characterisation of such **universal constructions** is a speciality of category theory; in our case, parallel composition can be shown to satisfy some **pullback properties** (Section 4.2).
- Universal constructions are unique up to isomorphism, so it is convenient to establish isomorphisms about universal constructions. The pullback properties of parallel composition can thus help to construct the ornamental conversion isomorphisms (Section 4.3.1) and the modularity isomorphisms (Section 4.3.2).

$$
\begin{tikzcd}
& \mathbb{O}\textsc{rn} \arrow[dl, bend right, "\textsc{RSem}"'] \arrow[d, "\textsc{Ind}"] & \\
\mathbb{F}\textsc{ref} \arrow[r, bend left, "\textsc{FRefF}"] \arrow[r, phantom, "\cong"] & \mathbb{F}\textsc{am} \arrow[l, bend left, "\textsc{FRefC}"] \arrow[r, bend left, "\textsc{Com}"] \arrow[r, bend right, "\textsc{FamF}"'] & \mathbb{F}\textsc{un}
\end{tikzcd}
$$

Figure 4.1 Categories and functors for the ornament–refinement framework.

4.1 Categories and functors

A first approximation of a category is a (directed multi-) **graph**, which consists of a set of objects (nodes) and a collection of sets of morphisms (edges) indexed with their source and target objects:

```
record Graph {l m : Level} : Set (suc (l ⊔ m)) where
  field
    Object : Set l
    _⇒_    : Object → Object → Set m
```

For example, the underlying graph of the category 𝔽UN of (small) sets and (total) functions is just

```
𝔽UN-graph : Graph
𝔽UN-graph = record { Object = Set
                   ; _⇒_    = λ A B ↦ A → B }
```

A category is a graph whose morphisms are equipped with a monoid-like compositional structure — there is a morphism composition operator of type

```
_·_ : {X Y Z : Object} → (Y ⇒ Z) → (X ⇒ Y) → (X ⇒ Z)
```

which has identities and is associative.

Syntactic remark (*universe polymorphism*). Many definitions in this chapter (like Graph above) employ Agda's universe polymorphism [Harper and Pollack, 1991], so the definitions can be instantiated at suitable levels of the Set hierarchy as needed. (For example, the type of $\mathbb{F}$UN-*graph* is implicitly instantiated as Graph $\{1\}$ $\{1\}$, since both Set and any $A \to B$ (where A, B : Set) are of type Set_1.) We will give the first few universe-polymorphic definitions with full detail about the levels, but will later suppress the syntactic noise wherever possible. □

Before we move on to the definition of categories, though, special attention must be paid to equality on morphisms, which is usually coarser than definitional equality — in $\mathbb{F}$UN, for example, it is necessary to identify functions up to extensional equality (so uniqueness of morphisms in universal properties would make sense). One ad hoc way to achieve this in Agda's intensional setting is to use **setoids** [Barthe et al., 2003] — sets with an explicitly specified equivalence relation — to represent sets of morphisms. The type of setoids can be defined as a record which contains a carrier set, an equivalence relation on the set, and the three laws for the equivalence relation:

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Setoid}\ \{c\ d\ :\ \mathsf{Level}\}\ :\ \mathsf{Set}\ (\mathsf{suc}\ (c \sqcup d))\ \textbf{where} \\
\quad \textbf{field} \\
\qquad \mathit{Carrier}\ :\ \mathsf{Set}\ c \\
\qquad \_{\approx}\_\ \ \ :\ \mathit{Carrier} \to \mathit{Carrier} \to \mathsf{Set}\ d \\
\qquad \mathit{refl}\ \ \ :\ \{x : \mathit{Carrier}\} \to x \approx x \\
\qquad \mathit{sym}\ \ \ :\ \{x\ y : \mathit{Carrier}\} \to x \approx y \to y \approx x \\
\qquad \mathit{trans}\ :\ \{x\ y\ z : \mathit{Carrier}\} \to x \approx y \to y \approx z \to x \approx z
\end{array}
$$

For example, we can define a setoid of functions that uses extensional equality:

$$
\begin{array}{l}
\mathit{FunSetoid}\ :\ \mathsf{Set} \to \mathsf{Set} \to \mathsf{Setoid} \\
\mathit{FunSetoid}\ A\ B\ =\ \textbf{record}\ \{\ \mathit{Carrier}\ =\ A \to B \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ \_{\approx}\_\ \ \ =\ \_{\doteq}\_ \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ \boxed{\text{proofs of laws}}\ \}
\end{array}
$$

Proofs of the three laws are omitted from the presentation.

Similarly, we can define the type of categories as a record containing a set of objects, a collection of **setoids** of morphisms indexed by source and target

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Category}\ \{l\ m\ n\ :\ \mathsf{Level}\}\ :\ \mathsf{Set}\ (\mathsf{suc}\ (l \sqcup m \sqcup n))\ \textbf{where} \\
\quad \textbf{field} \\
\qquad \mathit{Object} \quad\ \ :\ \mathsf{Set}\ l \\
\qquad \mathit{Morphism}\ :\ \mathit{Object} \to \mathit{Object} \to \mathsf{Setoid}\ \{m\}\ \{n\} \\
\quad \_{\Rightarrow}\_\ :\ \mathit{Object} \to \mathit{Object} \to \mathsf{Set}\ m \\
\quad X \Rightarrow Y\ =\ \mathsf{Setoid}.\mathit{Carrier}\ (\mathit{Morphism}\ X\ Y) \\
\quad \_{\approx}\_\ :\ \{X\ Y\ :\ \mathit{Object}\} \to (X \Rightarrow Y) \to (X \Rightarrow Y) \to \mathsf{Set}\ n \\
\quad \_{\approx}\_\ \{X\}\ \{Y\}\ =\ \mathsf{Setoid}.\_{\approx}\_\ (\mathit{Morphism}\ X\ Y) \\
\quad \textbf{field} \\
\qquad \_{\cdot}\_\ :\ \{X\ Y\ Z\ :\ \mathit{Object}\} \to (Y \Rightarrow Z) \to (X \Rightarrow Y) \to (X \Rightarrow Z) \\
\qquad \mathit{id} \quad :\ \{X\ :\ \mathit{Object}\} \to (X \Rightarrow X) \\
\qquad \mathit{id\text{-}l} \quad\ \ :\ \{X\ Y\ :\ \mathit{Object}\}\ (f\ :\ X \Rightarrow Y) \to \\
\qquad\qquad\quad\ \ \mathit{id} \cdot f\ \approx\ f \\
\qquad \mathit{id\text{-}r} \quad\ \ :\ \{X\ Y\ :\ \mathit{Object}\}\ (f\ :\ X \Rightarrow Y) \to \\
\qquad\qquad\quad\ \ f \cdot \mathit{id}\ \approx\ f \\
\qquad \mathit{assoc} \ \ :\ \{X\ Y\ Z\ W\ :\ \mathit{Object}\}\ (f\ :\ Z \Rightarrow W)\ (g\ :\ Y \Rightarrow Z)\ (h\ :\ X \Rightarrow Y) \to \\
\qquad\qquad\quad\ \ (f \cdot g) \cdot h\ \approx\ f \cdot (g \cdot h) \\
\qquad \mathit{cong\text{-}l}\ :\ \{X\ Y\ Z\ :\ \mathit{Object}\}\ \{f\ g\ :\ Y \Rightarrow Z\}\ (h\ :\ X \Rightarrow Y) \to \\
\qquad\qquad\quad\ \ f\ \approx\ g \to f \cdot h\ \approx\ g \cdot h \\
\qquad \mathit{cong\text{-}r}\ :\ \{X\ Y\ Z\ :\ \mathit{Object}\}\ (h\ :\ Y \Rightarrow Z)\ \{f\ g\ :\ X \Rightarrow Y\} \to \\
\qquad\qquad\quad\ \ f\ \approx\ g \to h \cdot f\ \approx\ h \cdot g
\end{array}
$$

Figure 4.2 Definition of categories.

objects, the composition operator on morphisms, the identity morphisms, and the identity and associativity laws for composition. The definition is shown in Figure 4.2. Two notations are introduced to improve readability: $X \Rightarrow Y$ is defined to be the carrier set of the setoid of morphisms from X to Y, and $f \approx g$ is defined to be the equivalence between the morphisms f and g as specified by the setoid to which f and g belong. The last two laws *cong-l* and *cong-r* require morphism composition to preserve the equivalence on morphisms; they are given in this form to work better with the equational reasoning combinators commonly used in Agda (see, for example, the AoPA library [Mu et al., 2009]).

Now we can define the category $\mathbb{F}\textsc{un}$ of sets and functions as

$$
\begin{array}{l}
\mathbb{F}\textsc{un} \;:\; \mathsf{Category} \\
\mathbb{F}\textsc{un} \;=\; \textbf{record}\; \{\; \mathit{Object} \;=\; \mathsf{Set} \\
\qquad\qquad\qquad\quad ;\; \mathit{Morphism} \;=\; \mathit{FunSetoid} \\
\qquad\qquad\qquad\quad ;\; \_\cdot\_ \;=\; \_\circ\_ \\
\qquad\qquad\qquad\quad ;\; \mathit{id} \;=\; \lambda\, x \mapsto x \\
\qquad\qquad\qquad\quad ;\; \boxed{\text{proofs of laws}}\;\}
\end{array}
$$

Another important category that we will make use of is $\mathbb{F}\textsc{am}$, the category of indexed families of sets and indexed families of functions, which is useful for talking about componentwise structures. An object in $\mathbb{F}\textsc{am}$ has type $\Sigma[\, I : \mathsf{Set}\,]\; I \to \mathsf{Set}$, i.e., it is a set I and a family of sets indexed by I; a morphism from $(J\,,\,Y)$ to $(I\,,\,X)$ is a function $e \,:\, J \to I$ and a family of functions from $Y\,j$ to $X\,(e\,j)$ for each $j \,:\, J$.

$$
\begin{array}{l}
\mathbb{F}\textsc{am} \;:\; \mathsf{Category} \\
\mathbb{F}\textsc{am} \;=\; \textbf{record} \\
\quad \{\; \mathit{Object} \;=\; \Sigma[\, I : \mathsf{Set}\,]\; I \to \mathsf{Set} \\
\quad ;\; \mathit{Morphism} \;=\; \lambda\,(J\,,\,Y)\;(I\,,\,X) \;\mapsto\; \textbf{record} \\
\qquad\qquad\qquad\quad \{\; \mathit{Carrier} \;=\; \Sigma[\, e : J \to I\,]\; Y \rightrightarrows (X \circ e) \\
\qquad\qquad\qquad\quad ;\; \_\approx\_ \;=\; \lambda\,(e\,,\,u)\;(e'\,,\,u') \;\mapsto \\
\qquad\qquad\qquad\qquad\qquad (e \doteq e') \times ((j \,:\, J) \to u\,\{j\} \approxeq u'\,\{j\}) \\
\qquad\qquad\qquad\quad ;\; \boxed{\text{proofs of laws}}\;\} \\
\quad ;\; \_\cdot\_ \;=\; \lambda\,(e\,,\,u)\;(f\,,\,v) \;\mapsto\; (e \circ f)\,,\,(\lambda\,\{k\} \;\mapsto\; u\,\{f\,k\} \circ v\,\{k\})
\end{array}
$$

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Functor}\ \{l\ m\ n\ l'\ m'\ n'\ :\ \mathsf{Level}\} \\
\quad (C\ :\ \mathsf{Category}\ \{l\}\ \{m\}\ \{n\})\ (D\ :\ \mathsf{Category}\ \{l'\}\ \{m'\}\ \{n'\})\ : \\
\quad \mathsf{Set}\ (l \sqcup m \sqcup n \sqcup l' \sqcup m' \sqcup n')\ \textbf{where} \\
\quad \textbf{field} \\
\quad\quad \mathit{object}\quad : \mathit{Object}\ C \to \mathit{Object}\ D \\
\quad\quad \mathit{morphism}\ :\ \{X\ Y\ :\ \mathit{Object}\ C\} \to X \Rightarrow_C Y \to \mathit{object}\ X \Rightarrow_D \mathit{object}\ Y \\
\quad\quad \mathit{equiv\text{-}preserving}\ :\ \{X\ Y\ :\ \mathit{Object}\ C\}\ \{f\ g\ :\ X \Rightarrow_C Y\} \to \\
\quad\quad\quad f\ \approx_C\ g \to \mathit{morphism}\, f\ \approx_D\ \mathit{morphism}\ g \\
\quad\quad \mathit{id\text{-}preserving}\quad :\ \{X\ :\ \mathit{Object}\ C\} \to \\
\quad\quad\quad \mathit{morphism}\ (\mathit{id}\ C\ \{X\})\ \approx_D\ \mathit{id}\ D\ \{\mathit{object}\ X\} \\
\quad\quad \mathit{comp\text{-}preserving}\ :\ \{X\ Y\ Z\ :\ \mathit{Object}\ C\}\ (f\ :\ Y \Rightarrow_C Z)\ (g\ :\ X \Rightarrow_C Y) \to \\
\quad\quad\quad \mathit{morphism}\ (f \cdot_C g)\ \approx_D\ (\mathit{morphism}\, f \cdot_D \mathit{morphism}\ g)
\end{array}
$$

Figure 4.3 Definition of functors. Subscripts are used to indicate to which category an operator belongs.

$$
\begin{array}{l}
\quad ;\ \mathit{id}\quad =\ (\lambda\, x\ \mapsto\ x)\ ,\ (\lambda\ \{i\}\ x\ \mapsto\ x) \\
\quad ;\ \boxed{\text{proofs of laws}}\ \}
\end{array}
$$

Note that the equivalence on morphisms is defined to be componentwise extensional equality, which is formulated with the help of McBride's "John Major" heterogeneous equality $\_\approxeq\_$ [McBride, 1999] — the equivalence $\_\dot{\approxeq}\_$ is defined by $g\ \dot{\approxeq}\ h\ =\ \forall\, x \to g\, x\ \approxeq\ h\, x$. (Given $y\ :\ Y\, j$ for some $j\ :\ J$, the types of $u\ \{j\}\ y$ and $u'\ \{j\}\ y$ are not definitionally equal but only provably equal, so it is necessary to employ heterogeneous equality.)

Categories are graphs with a compositional structure, and **functors** are transformations between categories that preserve the compositional structure. The definition of functors is shown in Figure 4.3: a functor consists of two mappings, one on objects and the other on morphisms, where the morphism part preserves all structures on morphisms, including equivalence, identity, and composition. For example, we have two functors from $\mathbb{F}$AM to $\mathbb{F}$UN, one summing components together

$$
\begin{aligned}
&\textsc{Com} \;:\; \mathsf{Functor}\;\mathbb{F}\textsc{am}\;\mathbb{F}\textsc{un} \quad \text{-- the comprehension functor}\\
&\textsc{Com} \;=\; \textbf{record}\;\{\; \mathit{object} \;=\; \lambda\,(I\,,X)\;\mapsto\;\Sigma\,I\,X\\
&\qquad\qquad\qquad\;;\; \mathit{morphism} \;=\; \lambda\,(e\,,u\,)\;\mapsto\; e * u\\
&\qquad\qquad\qquad\;;\; \boxed{\text{proofs of laws}}\;\}
\end{aligned}
$$

and the other extracting the index part.

$$
\begin{aligned}
&\textsc{FamF} \;:\; \mathsf{Functor}\;\mathbb{F}\textsc{am}\;\mathbb{F}\textsc{un} \quad \text{-- the family fibration functor}\\
&\textsc{FamF} \;=\; \textbf{record}\;\{\; \mathit{object} \;=\; \lambda\,(I\,,X)\;\mapsto\;I\\
&\qquad\qquad\qquad\;;\; \mathit{morphism} \;=\; \lambda\,(e\,,u\,)\;\mapsto\; e\\
&\qquad\qquad\qquad\;;\; \boxed{\text{proofs of laws}}\;\}
\end{aligned}
$$

The functor laws should be proved for both functors alongside their object and morphism maps. In particular, we need to prove that the morphism part preserves equivalence: for Com, this means we need to prove, for all $e : J \rightarrow I$, $u : Y \rightrightarrows (X \circ e)$ and $f : J \rightarrow I$, $v : Y \rightrightarrows (X \circ f)$, that

$$(e \doteq f) \times ((j : J) \rightarrow u\,\{j\} \stackrel{\cdot}{\approx} v\,\{j\}) \rightarrow (e * u \doteq f * v)$$

and for FamF we need to prove

$$(e \doteq f) \times ((j : J) \rightarrow u\,\{j\} \stackrel{\cdot}{\approx} v\,\{j\}) \rightarrow (e \doteq f)$$

both of which can be easily discharged.

Categories for refinements and ornaments

Some constructions in Chapter 3 can now be organised under several categories and functors. For a start, we already saw that refinements are interesting only because of their intensional contents; this is reflected in an isomorphism of categories between the category $\mathbb{F}$am and the category $\mathbb{F}$ref of type families and refinement families (i.e., there are two functors back and forth inverse to each other). An object in $\mathbb{F}$ref is an indexed family of sets as in $\mathbb{F}$am, and a morphism from $(J\,,Y)$ to $(I\,,X)$ consists of a function $e : J \rightarrow I$ on the indices and a refinement family of type $\mathsf{FRefinement}\;e\;X\;Y$. As for the equivalence on morphisms, it suffices to use extensional equality on the index functions and componentwise extensional equality on refinement families, where extensional equality on refinements means extensional equality on their forgetful functions (extracted by $\mathsf{Refinement}.\mathit{forget}$), which we have shown in Section 3.1.1 to be the core of refinements. Formally:

$$
\begin{array}{l}
\mathbb{F}\textsc{ref} \ : \ \mathsf{Category} \\
\mathbb{F}\textsc{ref} \ = \ \textbf{record} \\
\quad \{ \ \mathit{Object} \quad = \ \Sigma[\, I : \mathsf{Set} \,] \ I \to \mathsf{Set} \\
\quad ; \ \mathit{Morphism} \ = \ \lambda \, (J \, , Y) \ (I \, , X) \ \mapsto \ \textbf{record} \\
\qquad\qquad\qquad\qquad \{ \ \mathit{Carrier} \ = \ \Sigma[\, e : J \to I \,] \ \mathsf{FRefinement} \ e \ X \ Y \\
\qquad\qquad\qquad\qquad ; \ \_{\approx}\_ \ = \ \lambda \, (e \, , \mathit{rs}) \ (e' \, , \mathit{rs}') \ \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad (e \ \doteq \ e') \ \times \\
\qquad\qquad\qquad\qquad\qquad\qquad ((j \ : \ J) \to \mathsf{Refinement}.\mathit{forget} \ (\mathit{rs} \ \ (\mathsf{ok} \ j)) \ \stackrel{\sim}{\approx} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Refinement}.\mathit{forget} \ (\mathit{rs}' \ (\mathsf{ok} \ j))) \\
\qquad\qquad\qquad\qquad ; \ \text{proofs of laws} \ \} \\
\quad ; \ \text{proofs of laws} \ \}
\end{array}
$$

Note that a refinement family from $X \ : \ I \to \mathsf{Set}$ to $Y \ : \ J \to \mathsf{Set}$ is deliberately cast as a morphism in the opposite direction from $(J \, , Y)$ to $(I \, , X)$; think of this as suggesting the direction of the forgetful functions of refinements. $\mathbb{F}\textsc{ref}$ is no more powerful than $\mathbb{F}\textsc{am}$ since $\mathbb{F}\textsc{ref}$ ignores the intensional contents of refinements by using an extensional equality, and consequently there are two functors between $\mathbb{F}\textsc{ref}$ and $\mathbb{F}\textsc{am}$ that are inverse to each other, forming an isomorphism of categories:

- We have a forgetful functor $\mathrm{FR\textsc{ef}F} \ : \ \mathsf{Functor} \ \mathbb{F}\textsc{ref} \ \mathbb{F}\textsc{am}$ which is identity on objects and componentwise $\mathsf{Refinement}.\mathit{forget}$ on morphisms (which preserves equivalence automatically):

$$
\begin{array}{l}
\mathrm{FR\textsc{ef}F} \ : \ \mathsf{Functor} \ \mathbb{F}\textsc{ref} \ \mathbb{F}\textsc{am} \\
\mathrm{FR\textsc{ef}F} \ = \ \textbf{record} \\
\quad \{ \ \mathit{object} \qquad = \ \mathit{id} \\
\quad ; \ \mathit{morphism} \ = \ \lambda \, (e \, , \mathit{rs}) \ \mapsto \ e \, , (\lambda \, j \ \mapsto \ \mathsf{Refinement}.\mathit{forget} \ (\mathit{rs} \ (\mathsf{ok} \ j))) \\
\quad ; \ \text{proofs of laws} \ \}
\end{array}
$$

  Note that $\mathrm{FR\textsc{ef}F}$ remains a familiar covariant functor rather than a contravariant one because of our choice of morphism direction.

- Conversely, there is a functor FRefC : Functor $\mathbb{F}$am $\mathbb{F}$ref whose object part is identity and whose morphism part is componentwise *canonRef*:

$$
\begin{aligned}
&\text{FRefC} \;:\; \mathsf{Functor}\ \mathbb{F}\textsc{am}\ \mathbb{F}\textsc{ref} \\
&\text{FRefC} \;=\; \textbf{record} \\
&\quad \{\ \mathit{object} \quad\;\;\; = \mathit{id} \\
&\quad ;\ \mathit{morphism} \;=\; \lambda\,(e\,,u) \mapsto e\,,(\lambda\,(\mathsf{ok}\ j) \mapsto \mathit{canonRef}\ (u\ \{j\})) \\
&\quad ;\ \text{proofs of laws}\ \}
\end{aligned}
$$

The two functors FRefF and FRefC are inverse to each other by definition.

There is another category $\mathbb{O}$rn, which has objects of type $\Sigma[\,I : \mathsf{Set}\,]\ \mathsf{Desc}\ I$, i.e., descriptions paired with index sets, and morphisms from $(J\,,E)$ to $(I\,,D)$ of type $\Sigma[\,e : J \to I\,]\ \mathsf{Orn}\ e\ D\ E$, i.e., ornaments paired with index erasure functions. To complete the definition of $\mathbb{O}$rn:

- We need to devise an equivalence on ornaments

$$
\begin{aligned}
&\mathit{OrnEq} \;:\; \{I\ J \,:\, \mathsf{Set}\}\ \{e\ f \,:\, J \to I\}\ \{D \,:\, \mathsf{Desc}\ I\}\ \{E \,:\, \mathsf{Desc}\ J\} \to \\
&\qquad\qquad \mathsf{Orn}\ e\ D\ E \to \mathsf{Orn}\ f\ D\ E \to \mathsf{Set}
\end{aligned}
$$

  such that it implies extensional equality of e and f and that of ornamental forgetful functions:

$$
\begin{aligned}
&\mathit{OrnEq\text{-}forget} \;:\; \{I\ J \,:\, \mathsf{Set}\}\ \{e\ f \,:\, J \to I\}\ \{D \,:\, \mathsf{Desc}\ I\}\ \{E \,:\, \mathsf{Desc}\ J\} \to \\
&\qquad\qquad (O \,:\, \mathsf{Orn}\ e\ D\ E)\ (P \,:\, \mathsf{Orn}\ f\ D\ E) \to \mathit{OrnEq}\ O\ P \to \\
&\qquad\qquad (e \doteq f) \times ((j \,:\, J) \to \mathit{forget}\ O\ \{j\} \mathrel{\dot{\approx}} \mathit{forget}\ P\ \{j\})
\end{aligned}
$$

  The actual definition of *OrnEq* is deferred until Chapter 6.

- Morphism composition is sequential composition $\_\odot\_$, which merges two successive batches of modifications in a straightforward way. The definition is shown in Figure 4.4. There is also a family of **identity ornaments**:

$$
\begin{aligned}
&\mathit{idOrn} \;:\; \{I \,:\, \mathsf{Set}\}\ (D \,:\, \mathsf{Desc}\ I) \to \mathsf{Orn}\ \mathit{id}\ D\ D \\
&\mathit{idOrn}\ \{I\}\ D\ (\mathsf{ok}\ i) \;=\; \mathit{idROrn}\ (D\ i) \\
&\quad \textbf{where} \\
&\qquad \mathbb{E}\text{-}\mathit{refl} \;:\; (\mathit{is} \,:\, \mathsf{List}\ I) \to \mathbb{E}\ \mathit{id}\ \mathit{is}\ \mathit{is} \\
&\qquad \mathbb{E}\text{-}\mathit{refl}\ [\,] \qquad\;\; = [\,] \\
&\qquad \mathbb{E}\text{-}\mathit{refl}\ (i :: \mathit{is}) \;=\; \mathsf{refl} :: \mathbb{E}\text{-}\mathit{refl}\ \mathit{is}
\end{aligned}
$$

```
𝔼-trans : {I J K : Set} {e : J → I} {f : K → J} →
          {is : List I} {js : List J} {ks : List K} →
          𝔼 e js is → 𝔼 f ks js → 𝔼 (e ∘ f) ks is
𝔼-trans         []            []            = []
𝔼-trans {e := e} (eeq :: eeqs) (feq :: feqs) = trans (cong e feq) eeq ::
                                                𝔼-trans eeqs feqs

scROrn : {I J K : Set} {e : J → I} {f : K → J} →
         {D : RDesc I} {E : RDesc J} {F : RDesc K} →
         ROrn e D E → ROrn f E F → ROrn (e ∘ f) D F
scROrn (v eeqs)  (v feqs)  = v (𝔼-trans eeqs feqs)
scROrn (v eeqs)  (Δ T P)   = Δ[ t : T ] scROrn (v eeqs)  (P t)
scROrn (σ S O)   (σ .S P)  = σ[ s : S ] scROrn (O s)     (P s)
scROrn (σ S O)   (Δ T P)   = Δ[ t : T ] scROrn (σ S O)   (P t)
scROrn (σ S O)   (∇ s P)   = ∇[ s ]     scROrn (O s)     P
scROrn (Δ T O)   (σ .T P)  = Δ[ t : T ] scROrn (O t)     (P t)
scROrn (Δ T O)   (Δ U P)   = Δ[ u : U ] scROrn (Δ T O)   (P u)
scROrn (Δ T O)   (∇ t P)   =            scROrn (O t)     P
scROrn (∇ s O)   P         = ∇[ s ]     scROrn O         P

_⊙_ : {I J K : Set} {e : J → I} {f : K → J} →
      {D : Desc I} {E : Desc J} {F : Desc K} →
      Orn e D E → Orn f E F → Orn (e ∘ f) D F
_⊙_ {f := f} O P (ok k) = scROrn (O (ok (f k))) (P (ok k))
```

Figure 4.4 Definitions for sequential composition of ornaments.

$$\begin{aligned}
&\mathit{idROrn} \;:\; (E \;:\; \mathsf{RDesc}\; I) \to \mathsf{ROrn}\; \mathit{id}\; E\; E \\
&\mathit{idROrn}\; (\mathsf{v}\; \mathit{is}) \quad = \; \mathsf{v}\; (\mathbb{E}\text{-}\mathit{refl}\; \mathit{is}) \\
&\mathit{idROrn}\; (\sigma\; S\; E) \;=\; \sigma[\, s : S \,]\; \mathit{idROrn}\; (E\; s)
\end{aligned}$$

which simply use σ and v everywhere to express that a description is identical to itself. Unsurprisingly, the identity ornaments serve as identity of sequential composition.

To summarise:

$$\begin{aligned}
&\mathbb{O}\textsc{rn} \;:\; \mathsf{Category} \\
&\mathbb{O}\textsc{rn} \;=\; \textbf{record} \\
&\quad \{\; \mathit{Object} \quad = \Sigma[\, I : \mathsf{Set} \,]\; \mathsf{Desc}\; I \\
&\quad ;\; \mathit{Morphism} \;=\; \lambda\, (J\, ,\, E)\; (I\, ,\, D) \;\mapsto\; \textbf{record} \\
&\qquad\qquad\qquad\qquad \{\; \mathit{Carrier} \;=\; \Sigma[\, e : J \to I \,]\; \mathsf{Orn}\; e\; D\; E \\
&\qquad\qquad\qquad\qquad ;\; _{\approx}_ \quad = \; \lambda\, (e\, ,\, O)\; (f\, ,\, P) \;\mapsto\; \mathit{OrnEq}\; O\; P \\
&\qquad\qquad\qquad\qquad ;\; \boxed{\text{proofs of laws}}\; \} \\
&\quad ;\; _\cdot_ \;=\; \lambda\, (e\, ,\, O)\; (f\, ,\, P) \;\mapsto\; (e \circ f)\, ,\, (O \odot P) \\
&\quad ;\; \mathit{id} \quad = \; \lambda\, \{I\, ,\, D\} \;\mapsto\; \mathit{id}\, ,\, \mathit{idOrn}\; D \\
&\quad ;\; \boxed{\text{proofs of laws}}\; \}
\end{aligned}$$

A functor $\textsc{Ind} \;:\; \mathsf{Functor}\; \mathbb{O}\textsc{rn}\; \mathbb{F}\textsc{am}$ can then be constructed, which gives the ordinary semantics of descriptions and ornaments: the object part of $\textsc{Ind}$ decodes a description $(I\, ,\, D)$ to its least fixed point $(I\, ,\, \mu\, D)$, and the morphism part translates an ornament $(e\, ,\, O)$ to the forgetful function $(e\, ,\, \mathit{forget}\; O)$, the latter preserving equivalence by virtue of *OrnEq-forget*.

$$\begin{aligned}
&\textsc{Ind} \;:\; \mathsf{Functor}\; \mathbb{O}\textsc{rn}\; \mathbb{F}\textsc{am} \\
&\textsc{Ind} \;=\; \textbf{record}\; \{\; \mathit{object} \quad = \; \lambda\, (I\, ,\, D) \;\mapsto\; I\, ,\, \mu\, D \\
&\qquad\qquad\qquad ;\; \mathit{morphism} \;=\; \lambda\, (e\, ,\, O) \;\mapsto\; e\, ,\, \mathit{forget}\; O \\
&\qquad\qquad\qquad ;\; \boxed{\text{proofs of laws}}\; \}
\end{aligned}$$

To translate $\mathbb{O}\textsc{rn}$ to $\mathbb{F}\textsc{ref}$, i.e., datatype declarations to refinements, a naive way is to use the composite functor

$$\mathbb{O}\textsc{rn} \xrightarrow{\;\textsc{Ind}\;} \mathbb{F}\textsc{am} \xrightarrow{\;\textsc{FRefC}\;} \mathbb{F}\textsc{ref}$$

The resulting refinements would then use the canonical promotion predicates. However, the whole point of incorporating $\mathbb{O}\textsc{rn}$ in the framework is that we can construct an alternative functor RSEM directly from $\mathbb{O}\textsc{rn}$ to $\mathbb{F}\textsc{ref}$. The functor RSEM is extensionally equal to the above composite functor, but intensionally very different. While its object part still takes the least fixed point of a description, its morphism part is the refinement semantics of ornaments given in Section 3.3, whose promotion predicates are the optimised predicates and have a more efficient representation.

$$
\begin{aligned}
&\mathrm{RS}\textsc{em} \;:\; \mathsf{Functor}\ \mathbb{O}\textsc{rn}\ \mathbb{F}\textsc{am}\\
&\mathrm{RS}\textsc{em} \;=\; \textbf{record}\ \{\ \mathit{object} \;\;\;\;\; = \lambda\,(I\,,D) \mapsto I\,,\mu\,D\\
&\qquad\qquad\qquad\qquad ;\ \mathit{morphism} = \lambda\,(e\,,O) \mapsto e\,,\mathit{RSem}\ O\\
&\qquad\qquad\qquad\qquad ;\ \text{proofs of laws}\ \}
\end{aligned}
$$

Categorical isomorphisms

So far switching to the categorical language offers no obvious benefits.

Define the type of isomorphisms between two objects X and Y in C as

$$
\begin{aligned}
&\textbf{record}\ \mathsf{Iso}\ C\ X\ Y \;:\; \mathsf{Set}\ \_\ \textbf{where}\\
&\quad\textbf{field}\\
&\qquad \mathit{to} \quad\;\; : X \Rightarrow Y\\
&\qquad \mathit{from} \;: Y \Rightarrow X\\
&\qquad \mathit{from\text{-}to\text{-}inverse} \;:\; \mathit{from} \cdot \mathit{to} \approx \mathit{id}\\
&\qquad \mathit{to\text{-}from\text{-}inverse} \;:\; \mathit{to} \cdot \mathit{from} \approx \mathit{id}
\end{aligned}
$$

(The relation $\_\cong\_$ is formally defined as $\mathsf{Iso}\ \mathbb{F}\textsc{un}$.)

functors preserve isomorphisms (a quick demonstration of preorder reasoning); TBC

4.2 Pullback properties of parallel composition

One of the great advantages of category theory is the ability to formulate the idea of **universal constructions** generically and concisely, which we will use to give parallel composition a useful macroscopic characterisation. An intuitive way to understand the idea of a universal construction is to think of it as a "best" solution to some specification. More precisely, the specification is represented as a category whose objects are all possible solutions and whose morphisms are evidence of how the solutions "compare" with each other, and a "best" solution is a **terminal object** in this category, meaning that it is "evidently better" than all objects in the category. For the actual definition: an object in a category C is **terminal** when it satisfies the **universal property** that for every object X' there is a unique morphism from X' to X, i.e., the setoid *Morphism* X' X has a unique inhabitant: name scoping

$$\begin{aligned}
&\mathit{Terminal}\ C\ :\ \mathit{Object} \to \mathsf{Set}\ _\\
&\mathit{Terminal}\ C\ X\ =\ (X'\ :\ \mathit{Object}) \to \mathit{Singleton}\ (\mathit{Morphism}\ X'\ X)
\end{aligned}$$

where *Singleton* is defined by

$$\begin{aligned}
&\mathit{Singleton}\ :\ (S\ :\ \mathsf{Setoid}) \to \mathsf{Set}\ _\\
&\mathit{Singleton}\ S\ =\ \mathsf{Setoid}.\mathit{Carrier}\ S \times ((x\ y\ :\ \mathsf{Setoid}.\mathit{Carrier}\ S) \to x \approx_S y)
\end{aligned}$$

The uniqueness condition ensures that terminal objects are unique up to (a unique) isomorphism — that is, if two objects are both terminal in C, then there is an isomorphism between them:

$$\begin{aligned}
&\mathit{terminal\text{-}iso}\ C\ :\ (X\ Y\ :\ \mathit{Object}) \to \mathit{Terminal}\ C\ X \to \mathit{Terminal}\ C\ Y \to \mathsf{Iso}\ C\ X\ Y\\
&\mathit{terminal\text{-}iso}\ C\ X\ Y\ tX\ tY\ =\\
&\quad \mathbf{let}\ f\ :\ X \Rightarrow Y\\
&\qquad\ f\ =\ \mathsf{outl}\ (tY\ X)\\
&\qquad\ g\ :\ Y \Rightarrow X\\
&\qquad\ g\ =\ \mathsf{outl}\ (tX\ Y)\\
&\quad \mathbf{in}\ \ \mathbf{record}\ \{\ \mathit{to}\ =\ f\\
&\qquad\qquad\qquad ;\ \mathit{from}\ =\ g\\
&\qquad\qquad\qquad ;\ \mathit{from\text{-}to\text{-}inverse}\ =\ \mathsf{outr}\ (tX\ X)\ (g \cdot f)\ \mathit{id}
\end{aligned}$$

$$; \; \textit{to-from-inverse} \; = \; \mathsf{outr} \; (tY \; Y) \; (f \cdot g) \; \textit{id} \; \}$$

Thus, to prove that two constructions are isomorphic, one way would be to prove that they are universal in the same sense, i.e., they are both terminal objects in the same category. This is the main method we use to construct the ornamental conversion isomorphisms in Section 4.3.1 and the modularity isomorphisms in Section 4.3.2, both involving parallel composition. The goal of the rest of this section, then, is to find suitable universal properties that characterise parallel composition, preparing for Sections 4.3.1 and 4.3.2.

As said earlier, parallel composition computes the least informative ornament that contains the information of two compatible ornaments, and this is exactly a categorical **product**. Below we construct the definition of categorical products step by step. Let C be a category and L, R two objects in C. A **span** over L and R is defined by

```
record Span C L R : Set _ where
  constructor span
  field
    M : Object
    l : M ⇒ L
    r : M ⇒ R
```

or diagrammatically:

$$L \xleftarrow{\;l\;} M \xrightarrow{\;r\;} R$$

If we interpret a morphism $X \Rightarrow Y$ as evidence that X is more informative than Y, then a span over L and R is essentially an object which is more informative than both L and R. Spans over the same objects can be "compared": define a morphism between two spans by

```
record SpanMorphism C L R (s s' : Span C L R) : Set _ where
  constructor spanMorphism
  field
    m : Span.M s ⇒ Span.M s'
    triangle-l : Span.l s' · m ≈ Span.l s
    triangle-r : Span.r s' · m ≈ Span.r s
```

or diagrammatically (abbreviating $\mathsf{Span}.l\ s'$ to l' and so forth):

$$
\begin{tikzcd}
 & M \arrow[dl, "l"'] \arrow[dr, "r"] \arrow[dd, "m"] & \\
L & & R \\
 & M' \arrow[ul, "l'"] \arrow[ur, "r'"'] &
\end{tikzcd}
$$

where the two triangles are required to commute. Thus a span s is more informative than another span s' when $\mathsf{Span}.M\ s$ is more informative than $\mathsf{Span}.M\ s'$ and the morphisms factorise appropriately. We can then form a category of spans over L and R:

diagram commutativity not yet defined

morphism equivalence and proof irrelevance

$$
\begin{array}{l}
\mathit{SpanCategory}\ C\ L\ R\ :\ \mathsf{Category} \\
\mathit{SpanCategory}\ C\ L\ R\ =\ \mathbf{record} \\
\quad \{\ \mathit{Object}\quad =\ \mathsf{Span}\ C\ L\ R \\
\quad ;\ \mathit{Morphism}\ = \\
\qquad \lambda\, s\ s'\ \mapsto\ \mathbf{record} \\
\qquad\quad \{\ \mathit{Carrier}\ =\ \mathsf{SpanMorphism}\ C\ L\ R\ s\ s' \\
\qquad\quad ;\ \_{\approx}\_\ =\ \lambda\, f\ g\ \mapsto\ \mathsf{SpanMorphism}.m\ f\ \approx\ \mathsf{SpanMorphism}.m\ g \\
\qquad\quad ;\ \boxed{\text{proofs of laws}}\ \} \\
\quad ;\ \boxed{\text{proofs of laws}}\ \}
\end{array}
$$

and a product of L and R is a terminal object in this category:

$$
\begin{array}{l}
\mathit{Product}\ C\ L\ R\ :\ \mathsf{Span}\ C\ L\ R \to \mathsf{Set}\ \_ \\
\mathit{Product}\ C\ L\ R\ =\ \mathit{Terminal}\ (\mathit{SpanCategory}\ C\ L\ R)
\end{array}
$$

In particular, a product of L and R contains the least informative object that is more informative than both L and R.

product diagram; "morphism relevance"?

We thus aim to characterise parallel composition as a product of two compatible ornaments. This means that ornaments should be the objects of some category, but so far we only know that ornaments are morphisms of the category Orn. We are thus directed to construct a category whose objects are morphisms in an ambient category C, so when we use Orn as the ambient category, parallel composition can be characterised as a product in the derived category. Such a category is in general a **comma category** [Mac Lane, 1998, § II.6], whose objects are morphisms with arbitrary source and target objects, but here we should restrict ourselves to a special case called a **slice category**, since we seek to form products of only compatible ornaments (whose less informative end coincide) rather than arbitrary ones. A slice category is parametrised with an ambient category C and an object B in C, and has

- objects: all the morphisms in C with target B,

```
record Slice C B : Set _ where
  constructor slice
  field
    T : Object
    s : T ⇒ B
```

  and

- morphisms: mediating morphisms giving rise to commutative triangles,

```
record SliceMorphism C B (s s' : Slice C B) : Set _ where
  constructor sliceMorphism
  field
    m : Slice.T s ⇒ Slice.T s'
    triangle : Slice.s s' · m ≈ Slice.s s
```

or diagrammatically:

$$\text{objects}\quad \begin{tikzcd} T \arrow[d, "s"'] \\ B \end{tikzcd} \qquad \text{and} \qquad \text{morphisms}\quad \begin{tikzcd} T \arrow[rr, "m"] \arrow[dr, "s"'] & & T' \arrow[dl, "s'"] \\ & B & \end{tikzcd}$$

The definitions above are assembled into the definition of slice categories in much the same way as span categories:

```
SliceCategory C B : Category
SliceCategory C B = record
  { Object    = Slice C B
  ; Morphism  =
      λ s s' ↦ record
```

$$
\begin{aligned}
&\quad\{\ \mathit{Carrier}\ =\ \mathsf{SliceMorphism}\ C\ B\ s\ s'\\
&\quad;\ \_{\approx}\_\ =\ \lambda f\ g \mapsto \mathsf{SliceMorphism}.m\ f\ \approx\ \mathsf{SliceMorphism}.m\ g\\
&\quad;\ \text{proofs of laws}\ \}\\
&;\ \text{proofs of laws}\ \}
\end{aligned}
$$

Objects in a slice category are thus morphisms with a common target, and when the ambient category is $\mathbb{O}$RN, they are exactly the compatible ornaments that can be composed in parallel.

We have arrived at the characterisation of parallel composition as a product in a slice category on top of $\mathbb{O}$RN. The composite term "product in a slice category" has become a multi-layered concept and can be confusing; to facilitate comprehension, we give several new definitions that can sometimes deliver better intuition. Let C be an ambient category and X an object in C. We refer to spans over two slices $f, g\ :\ \mathsf{Slice}\ C\ X$ alternatively as **squares** over f and g:

$$
\begin{aligned}
&\mathsf{Square}\ C\ f\ g\ :\ \mathsf{Set}\ \_\\
&\mathsf{Square}\ C\ f\ g\ =\ \mathsf{Span}\ (\mathit{SliceCategory}\ C\ X)\ f\ g
\end{aligned}
$$

since diagrammatically a square looks like

$$
\begin{tikzcd}
 & W \arrow[dl, "l"'] \arrow[dr, "r"] \arrow[d] & \\
Y \arrow[r, "f"'] & X & Z \arrow[l, "g"]
\end{tikzcd}
\qquad \text{which is the same as} \qquad
\begin{tikzcd}
\overset{f}{Y} \arrow[d] & W \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & \overset{g}{Z} \arrow[d] \\
X \arrow[r, phantom, "="] & X \arrow[r, phantom, "="] & X
\end{tikzcd}
$$

In a square q, we will refer to the object $\mathsf{Slice}.T\ (\mathsf{Span}.M\ q)$, i.e., the node W in the diagrams above, as the **vertex** of q:

$$
\begin{aligned}
&\mathit{vertex}\ :\ \mathsf{Square}\ C\ f\ g \to \mathit{Object}\\
&\mathit{vertex}\ =\ \mathsf{Slice}.T \circ \mathsf{Span}.M
\end{aligned}
$$

A product of f and g is alternatively referred to as a **pullback** of f and g; that is, it is a square over f and g satisfying

$$
\begin{aligned}
&\mathit{Pullback}\ C\ f\ g\ :\ \mathsf{Square}\ C\ f\ g \to \mathsf{Set}\ \_\\
&\mathit{Pullback}\ C\ f\ g\ =\ \mathit{Product}\ (\mathit{SliceCategory}\ C\ X)\ f\ g
\end{aligned}
$$

Equivalently, if we define the **square category** over f and g as

$$\begin{aligned}
&\mathit{SquareCategory}\ \mathcal{C}\ f\ g\ :\ \mathsf{Category}\\
&\mathit{SquareCategory}\ \mathcal{C}\ f\ g\ =\ \mathit{SpanCategory}\ (\mathit{SliceCategory}\ \mathcal{C}\ X)\ f\ g
\end{aligned}$$

then a pullback of f and g is a terminal object in the square category over f and g — indeed, $\mathit{Product}\ (\mathit{SliceCategory}\ \mathcal{C}\ X)\ f\ g$ is definitionally equal to $\mathit{Terminal}\ (\mathit{SquareCategory}\ \mathcal{C}\ f\ g)$. This means that, by *terminal-iso*, there is an isomorphism between any two pullbacks p and q of the same slices f and g:

diagram of pullback

$$\mathsf{Iso}\ (\mathit{SquareCategory}\ \mathcal{C}\ f\ g)\ p\ q$$

Subsequently, since there is a forgetful functor from $\mathit{SquareCategory}\ \mathcal{C}\ f\ g$ to $\mathcal{C}$ whose object part is *vertex*, and functors preserve isomorphisms, we also have an isomorphism

$$\mathsf{Iso}\ \mathcal{C}\ (\mathit{vertex}\ p)\ (\mathit{vertex}\ q) \tag{4.1}$$

which is what we actually use in Sections 4.3.1 and 4.3.2.

pullback diagram; pullback preservation

We are now ready to state precisely the pullback properties for parallel composition that we make use of later. We could attempt to establish that, for any two ornaments $O\ :\ \mathsf{Orn}\ e\ D\ E$ and $P\ :\ \mathsf{Orn}\ f\ D\ F$ where $D\ :\ \mathsf{Desc}\ I$, $E\ :\ \mathsf{Desc}\ J$, and $F\ :\ \mathsf{Desc}\ K$, the following square in $\mathbf{Orn}$ is a pullback:

$$\begin{tikzcd}
e \bowtie f\ ,\ \lfloor O \otimes P \rfloor \arrow[rr, "\mathsf{outr}\ ,\ \mathit{diffOrn\text{-}r}\ O\ P"] \arrow[d, "\mathsf{outl}\ ,\ \mathit{diffOrn\text{-}l}\ O\ P"'] \arrow[drr, "\mathit{pull},\ \lceil O \otimes P \rceil" description] & & K\ ,\ F \arrow[d, "f\ ,\ P"] \\
J\ ,\ E \arrow[rr, "e\ ,\ O"'] & & I\ ,\ D
\end{tikzcd} \tag{4.2}$$

The Agda term for this square is

$$\begin{aligned}
&\mathit{pc\text{-}square}\ :\ \mathsf{Square}\ \mathbf{Orn}\ (\mathsf{slice}\ (J\ ,\ E)\ (e\ ,\ O))\ (\mathsf{slice}\ (K\ ,\ F)\ (f\ ,\ P))\\
&\mathit{pc\text{-}square}\ =\ \mathsf{span}\ (\mathsf{slice}\ (e \bowtie f\ ,\ \lfloor O \otimes P \rfloor)\ (\mathit{pull},\ \lceil O \otimes P \rceil))\\
&\qquad\qquad\qquad (\mathsf{sliceMorphism}\ (\mathsf{outl}\ ,\ \mathit{diffOrn\text{-}l}\ O\ P)\ \{\ \}_0)\\
&\qquad\qquad\qquad (\mathsf{sliceMorphism}\ (\mathsf{outr}\ ,\ \mathit{diffOrn\text{-}r}\ O\ P)\ \{\ \}_1)
\end{aligned}$$

where Goal 0 has type $\mathit{OrnEq}\ (O \odot \mathit{diffOrn\text{-}l}\ O\ P)\ \lceil O \otimes P \rceil$ and Goal 1 has type

OrnEq $(P \odot \textit{diffOrn-r}\ O\ P)\ \lceil O \otimes P \rceil$, both of which can be discharged. Comparing the commutative diagram (4.2) and the Agda term *pc-square*, it should be obvious how concise the categorical language can be — the commutative diagram expresses the structure of the Agda term in a clean and visually intuitive way. Since terms like *pc-square* can be reconstructed from commutative diagrams and the categorical definitions, from now on we will present commutative diagrams as representations of the corresponding Agda terms and omit the latter. The pullback property of (4.2) is not too useful by itself, though: $\mathbb{O}$RN is a quite restricted category, so a universal property established in $\mathbb{O}$RN has limited applicability. Instead, we are more interested in the pullback property of the image of (4.2) under IND in $\mathbb{F}$AM:

$$\begin{tikzcd} e \bowtie f\ ,\ \mu\ \lfloor O \otimes P \rfloor \arrow[r, "\mathsf{outr}\ ,\ \textit{forget}\ (\textit{diffOrn-r}\ O\ P)"] \arrow[d, "\mathsf{outl}\ ,\ \textit{forget}\ (\textit{diffOrn-l}\ O\ P)"'] \arrow[dr, "\textit{pull},\ \textit{forget}\ \lceil O \otimes P \rceil" description] \arrow[dr, phantom, very near start, "\lrcorner"] & K\ ,\ \mu\ F \arrow[d, "f\ ,\ \textit{forget}\ P"] \\ J\ ,\ \mu\ E \arrow[r, "e\ ,\ \textit{forget}\ O"'] & I\ ,\ \mu\ D \end{tikzcd} \tag{4.3}$$

We assert that the above square is a pullback by marking its vertex with "$\lrcorner$". The proof of its universal property boils down to, very roughly speaking, datatype-generic construction of an inverse to

$\textit{forget}\ (\textit{diffOrn-l}\ O\ P) \vartriangle \textit{forget}\ (\textit{diffOrn-r}\ O\ P)$

which involves tricky manipulation of equality proofs but is achievable. After the pullback square (4.3) is established in $\mathbb{F}$AM, since the functor COM is pullback-preserving, we also get a pullback square in $\mathbb{F}$UN:

$$\begin{tikzcd} \Sigma\ (e \bowtie f)\ (\mu\ \lfloor O \otimes P \rfloor) \arrow[r, "\mathsf{outr} * \textit{forget}\ (\textit{diffOrn-r}\ O\ P)"] \arrow[d, "\mathsf{outl} * \textit{forget}\ (\textit{diffOrn-l}\ O\ P)"'] \arrow[dr, "\textit{pull} * \textit{forget}\ \lceil O \otimes P \rceil" description] \arrow[dr, phantom, very near start, "\lrcorner"] & \Sigma\ K\ (\mu\ F) \arrow[d, "f * \textit{forget}\ P"] \\ \Sigma\ J\ (\mu\ E) \arrow[r, "e * \textit{forget}\ O"'] & \Sigma\ I\ (\mu\ D) \end{tikzcd} \tag{4.4}$$

mention commutativity, associativity; should probably refactor the pullback square in $\mathbb{F}$AM into the pullback square in ORN and pullback preservation of IND

4.3 Consequences

4.3.1 The ornamental conversion isomorphisms

We restate the ornamental conversion isomorphisms as follows: for any ornament O : $\mathsf{Orn}\ e\ D\ E$ where D : $\mathsf{Desc}\ I$ and E : $\mathsf{Desc}\ J$, we have

$$\mu\ E\ j \;\cong\; \Sigma[\, x : \mu\ D\ (e\ j)\,]\ \ \mathsf{OptP}\ O\ (\mathsf{ok}\ j)\ x$$

for all j : J. Since the optimised predicates $\mathsf{OptP}\ O$ are defined by parallel composition of O and the singleton ornament $S = \mathit{singletonOD}\ D$, the isomorphism expands to

$$\mu\ E\ j \;\cong\; \Sigma[\, x : \mu\ D\ (e\ j)\,]\ \ \mu\ \lfloor O \otimes \lceil S \rceil \rfloor\ (\mathsf{ok}\ j\ ,\ \mathsf{ok}\ (e\ j\ ,\ x)) \tag{4.5}$$

How do we derive this from the pullback properties for parallel composition? It turns out that the pullback property in $\mathbb{F}$UN (4.4) can help.

- First, observe that we have the following pullback square:

$$\begin{tikzcd}
\Sigma\ J\ (\mu\ E) \arrow[rr, "(e * \mathit{forget}\ O)\ \triangle\ (\mathit{singleton} \circ \mathit{forget}\ O \circ \mathsf{outr})"] \arrow[d, "\mathit{id}"'] \arrow[drr, "e * \mathit{forget}\ O"] \arrow[dr, phantom, "\lrcorner", very near start] & & \Sigma\ (\Sigma\ I\ (\mu\ D))\ (\mu\ \lfloor S \rfloor) \arrow[d, "\mathsf{outl} * \mathit{forget}\ \lceil S \rceil"] \\
\Sigma\ J\ (\mu\ E) \arrow[rr, "e * \mathit{forget}\ O"'] & & \Sigma\ I\ (\mu\ D)
\end{tikzcd} \tag{4.6}$$

  Viewing pullbacks as products of slices, since a singleton ornament does not add information to a datatype, the vertical slice on the right-hand side

$$s \;=\; \mathsf{slice}\ (\Sigma\ (\Sigma\ I\ (\mu\ D))\ (\mu\ \lfloor S \rfloor))\ (\mathsf{outl} * \mathit{forget}\ \lceil S \rceil)$$

  behaves like a "multiplicative unit": any (compatible) slice s' alone gives rise to a product of s and s'. As a consequence, we have the bottom-left type

$\Sigma\, J\ (\mu\ E)$ as the vertex of the pullback. This pullback square is over the same slices as the pullback square (4.4) with P substituted by $\lceil S \rceil$, so by (4.1) we obtain an isomorphism

$$\Sigma\, J\ (\mu\ E)\ \cong\ \Sigma\ (e \bowtie \mathsf{outl})\ (\mu\ \lfloor O \otimes \lceil S \rceil \rfloor) \tag{4.7}$$

- To get from (4.7) to (4.5), we need to look more closely into the construction of (4.7). The right-to-left direction of (4.7) is obtained by applying the universal property of (4.6) to the square (4.4) (with P substituted by $\lceil S \rceil$), so it is the unique mediating morphism m that makes the following diagram commute:

$$\begin{tikzcd}
 & \Sigma\ (e \bowtie \mathsf{outl})\ (\mu\ \lfloor O \otimes \lceil S \rceil \rfloor) \arrow[dl, "\mathsf{outl} * \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)"'] \arrow[dr, "\mathsf{outr} * \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ P)"] \arrow[dd, dashed, "m"'] & \\
\Sigma\, J\ (\mu\ E) & & \Sigma\ (\Sigma\, I\ (\mu\ D))\ (\mu\ \lfloor S \rfloor) \\
 & \Sigma\, J\ (\mu\ E) \arrow[ul, "\mathit{id}"] \arrow[ur, "(e * \mathit{forget}\ O)\ \triangle\ (\mathit{singleton} \circ \mathit{forget}\ O \circ \mathsf{outr})"'] &
\end{tikzcd}$$

From the left commuting triangle, we see that, extensionally, the morphism m is just $\mathsf{outl} * \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)$.

- This leads us to the following general lemma: if there is an isomorphism

$$\Sigma\, K\, X\ \cong\ \Sigma\, L\, Y$$

whose right-to-left direction is extensionally equal to some $f * g$, then we have

$$X\, k\ \cong\ \Sigma [\, l : f^{\,-1}\ k\,]\ \ Y\ (\mathit{und}\ l)$$

for all $k : K$. For a justification: fixing $k : K$, an element of the form $(k\ ,\ x) : \Sigma\ K\ X$ must correspond, under the given isomorphism, to some element $(l\ ,\ y) : \Sigma\ L\ Y$ such that $f\ l \equiv k$, so the set $X\ k$ corresponds to exactly the sum of the sets $Y\ l$ such that $f\ l \equiv k$.

- Specialising the lemma above for (4.7), we get

$$\mu\ E\ j\ \cong\ \Sigma [\, \mathit{jix} : \mathsf{outl}^{\,-1}\ j\,]\ \ \mu\ \lfloor O \otimes \lceil S \rceil \rfloor\ (\mathit{und}\ \mathit{jix}) \tag{4.8}$$

for all $j : J$. Finally, observe that a canonical element of type $\mathsf{outl}^{-1}\ j$ must be of the form $\mathsf{ok}\ (\mathsf{ok}\ j\ ,\ \mathsf{ok}\ (e\ j\ ,\ x))$ for some $x : \mu\ D\ (e\ j)$, so we perform a change of variables for the summation, turning the right-hand side of (4.8) into

$$\Sigma[\,x : \mu\ D\ (e\ j)\,]\ \ \mu\ \lfloor O \otimes \lceil S \rceil \rfloor\ (\mathsf{ok}\ j\ ,\ \mathsf{ok}\ (e\ j\ ,\ x))$$

and arriving at (4.5).

Formalisation detail. There is a twist when it comes to formalisation of the proof in Agda, however, due to Agda's intensionality: It is possible to formalise the lemma and the change of variables individually and chain them together, but the resulting isomorphisms would have a very complicated definition due to suspended type casts. If we use them to construct the refinement family in the morphism part of RSem, it would be rather difficult to prove that the morphism part of RSem preserves equivalence. We are thus forced to fuse all the above reasoning into one step to get a clean Agda definition such that RSem preserves equivalence automatically, but the idea is still essentially the same. □

4.3.2 The modularity isomorphisms

The other important family of isomorphisms we should construct from the pullback properties of parallel composition is the modularity isomorphisms, which is restated as follows: Suppose that there are descriptions $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$ and $F : \mathsf{Desc}\ K$, and ornaments $O : \mathsf{Orn}\ e\ D\ E$, and $P : \mathsf{Orn}\ f\ D\ F$. Then we have

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

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

$$\begin{aligned} &\mu\ \lfloor \lceil O \otimes P \rceil \otimes \lceil S \rceil \rfloor\ (\mathsf{ok}\ (j\ ,\ k)\ ,\ \mathsf{ok}\ (i\ ,\ x)) \\ &\quad \cong\ \mu\ \lfloor O \otimes \lceil S \rceil \rfloor\ (j\ ,\ \mathsf{ok}\ (i\ ,\ x)) \times \mu\ \lfloor P \otimes \lceil S \rceil \rfloor\ (k\ ,\ \mathsf{ok}\ (i\ ,\ x)) \end{aligned} \tag{4.9}$$

where again $S = \mathit{singletonOD}\ D$. A quick observation is that they are componentwise isomorphisms between the two families of sets

$$M \;=\; \mu \; \lfloor \lceil O \otimes P \rceil \otimes \lceil S \rceil \rfloor$$

and

$$\begin{aligned} N \;=\; & \lambda\,(\mathsf{ok}\;(j\,,k)\,,\mathsf{ok}\;(i\,,x)) \;\mapsto \\ & \quad \mu\;\lfloor O \otimes \lceil S \rceil \rfloor\;(j\,,\mathsf{ok}\;(i\,,x)) \times \mu\;\lfloor P \otimes \lceil S \rceil \rfloor\;(k\,,\mathsf{ok}\;(i\,,x)) \end{aligned}$$

both indexed by $\mathit{pull} \bowtie \mathsf{outl}$ where pull has type $e \bowtie f \to I$ and outl has type $\Sigma\; I\; X \to I$. This is just an isomorphism in $\mathbb{F}$AM between $(\mathit{pull} \bowtie \mathsf{outl}\;, M)$ and $(\mathit{pull} \bowtie \mathsf{outl}\;, N)$ whose index part (i.e., the isomorphism obtained under the functor FAMF) is identity. Thus we seek to prove that both $(\mathit{pull} \bowtie \mathsf{outl}\;, M)$ and $(\mathit{pull} \bowtie \mathsf{outl}\,, N)$ are vertices of pullbacks of the same slices.

- We look at $(\mathit{pull} \bowtie \mathsf{outl}\,, N)$ first. For fixed i, j, k, and x, the set

  $$N\;(\mathsf{ok}\;(j\,,k)\,,\mathsf{ok}\;(i\,,x))$$

  along with the cartesian projections is a product, which trivially extends to a pullback since there is a forgetful function from each of the two component sets to the **singleton** set $\mu\;\lfloor S \rfloor\;(i\,,x)$, as shown in the following diagram:

  $$\begin{tikzcd}[column sep=huge, row sep=huge]
N\;(\mathsf{ok}\;(j\,,k)\,,\mathsf{ok}\;(i\,,x)) \arrow[r, "\mathsf{outr}"] \arrow[d, "\mathsf{outl}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mu\;\lfloor P \otimes \lceil S \rceil \rfloor\;(k\,,\mathsf{ok}\;(i\,,x)) \arrow[d, "\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;P\;\lceil S \rceil)"] \\
\mu\;\lfloor O \otimes \lceil S \rceil \rfloor\;(j\,,\mathsf{ok}\;(i\,,x)) \arrow[r, "\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;\lceil S \rceil)"'] & \mu\;\lfloor S \rfloor\;(i\,,x)
\end{tikzcd}$$

  Note that this pullback square is possible because of the common x in the indices of the two component sets — otherwise they cannot project to the same singleton set. Collecting all such pullback squares together, we get the following pullback square in $\mathbb{F}$AM:

  $$\begin{tikzcd}[column sep=huge, row sep=huge]
\mathit{pull} \bowtie \mathsf{outl}\,, N \arrow[r, "_\,,\mathsf{outr}"] \arrow[d, "_\,,\mathsf{outl}"'] \arrow[dr, phantom, "\lrcorner", very near start] & f \bowtie \mathsf{outl}\;, \mu\;\lfloor P \otimes \lceil S \rceil \rfloor \arrow[d, "\mathsf{outr}\;,\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;P\;\lceil S \rceil)"] \\
e \bowtie \mathsf{outl}\;, \mu\;\lfloor O \otimes \lceil S \rceil \rfloor \arrow[r, "\mathsf{outr}\;,\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;\lceil S \rceil)"'] & \Sigma\; I\;(\mu\; D)\;, \mu\;\lfloor S \rfloor
\end{tikzcd} \tag{4.10}$$

- Next we prove that $(pull \bowtie \mathsf{outl}\ ,\ M)$ is also the vertex of a pullback of the same slices as (4.10). This second pullback arises as a consequence of the following lemma (illustrated in the diagram below): In any category, consider the objects X, Y, their product $X \Leftarrow X \boxtimes Y \Rightarrow Y$, and products of each of the three objects X, Y, and $X \boxtimes Y$ with an object Z. (All the projections are shown as solid arrows in the diagram.) Then $(X \boxtimes Y) \boxtimes Z$ is the vertex of a pullback of the two projections $X \boxtimes Z \Rightarrow Z$ and $Y \boxtimes Z \Rightarrow Z$.

$$
\begin{tikzcd}
X & & X \boxtimes Y \arrow[ll] \arrow[rr] & & Y \\
& & (X \boxtimes Y) \boxtimes Z \arrow[u] \arrow[dl, dashed] \arrow[dr, dashed] \arrow[dd] \arrow[d, phantom, very near start, "\vee"] & & \\
& X \boxtimes Z \arrow[uul] \arrow[dr] & {} & Y \boxtimes Z \arrow[uur] \arrow[dl] & \\
& & Z & &
\end{tikzcd}
$$

  We again intend to view a pullback as a product of slices, and instantiate the lemma in *SliceCategory* $\mathbb{F}$AM $(I\ ,\ \mu\ D)$, substituting all the objects by slices consisting of relevant ornamental forgetful functions in (4.9). The substitutions are as follows:

$$
\begin{aligned}
X &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ O) \\
Y &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ P) \\
X \boxtimes Y &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ \lceil O \otimes P \rceil) \\
Z &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ \lceil S \rceil) \\
X \boxtimes Z &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ \lceil O \otimes \lceil S \rceil \rceil) \\
Y \boxtimes Z &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ \lceil P \otimes \lceil S \rceil \rceil) \\
(X \boxtimes Y) \boxtimes Z &\mapsto \mathsf{slice}\ \_\ (\_\ , \mathit{forget}\ \lceil \lceil O \otimes P \rceil \otimes \lceil S \rceil \rceil)
\end{aligned}
$$

  where $X \boxtimes Y$, $X \boxtimes Z$, $Y \boxtimes Z$, and $(X \boxtimes Y) \boxtimes Z$ indeed give rise to products in *SliceCategory* $\mathbb{F}$AM $(I\ ,\ \mu\ D)$, i.e., pullbacks in $\mathbb{F}$AM, by instantiating (4.3). What we get out of this instantiation of the lemma is a pullback in *SliceCategory* $\mathbb{F}$AM $(I\ ,\ \mu\ D)$ rather than $\mathbb{F}$AM. This is easy to fix, since there is a forgetful functor from any *SliceCategory* $C\ B$ to C whose object part is $\mathsf{Slice}.T$, and it is pullback-preserving. We thus get a pullback in $\mathbb{F}$AM of the same slices as (4.10) whose vertex is $(pull \bowtie \mathsf{outl}\ , M)$.

Having the two pullbacks, by (4.1) we get an isomorphism in $\mathbb{F}\textsc{am}$ between $(\mathit{pull} \bowtie \mathsf{outl}\ ,\ M)$ and $(\mathit{pull} \bowtie \mathsf{outl}\ ,\ N)$, whose index part can be shown to be identity, so there are componentwise isomorphisms between M and N in $\mathbb{F}\textsc{un}$, arriving at (4.9).

4.4 Discussion

elimination of arbitrariness of type-theoretic constructions; functor-level abstraction; compare with purely categorical approach

Chapter 5

Relational algebraic ornaments

the synthetic direction of the conversion isomorphism; emphasis no longer only on program derivation (relational calculus) but also on relational specifications

5.1 Relational programming in Agda

intro needs revision to de-emphasise program derivation a bit

One common approach to program derivation is by algebraic transformations of functional programs: one begins with a specification in the form of a functional program that expresses straightforward but possibly inefficient computation, and transforms it into an extensionally equal but more efficient functional program by applying algebraic laws and theorems. Using functional programs as the specification language means that specifications are directly executable, but the deterministic nature of functional programs can result in less flexible specifications. For example, when specifying an optimisation problem using a functional program that generates all feasible solutions and chooses an optimal one among them, the program would enforce a particular way of choosing the optimal solution, but such enforcement should not be part of the specification. To gain more flexibility, the specification language was later generalised to **relational programs**. With relational programs, we specify only the relationship between input and output without actually specifying a way to execute the programs, so specifications in the form of relational programs can be as flexible as possible. Though lacking a directly executable semantics, most relational programs can still be read computationally as potentially partial and nondeterministic mappings, so relational specifications largely remain computationally intuitive as functional specifications.

To emphasise the computational interpretation of relations, we will mainly model a relation between sets A and B as a function sending each element of A to a **subset** of B. We define subsets by

$$\begin{aligned}&\mathcal{P} \;:\; \mathsf{Set} \to \mathsf{Set}_1\\ &\mathcal{P}\,A \;=\; A \to \mathsf{Set}\end{aligned}$$

That is, a subset $s : \mathcal{P}\,A$ is a characteristic function that assigns a type to each element of A, and $a : A$ is considered to be a member of s if the type $s\ a : \mathsf{Set}$ is inhabited. We may regard $\mathcal{P}\,A$ as the type of computations that nondeterministically produce an element of A. A simple example is

$$\begin{aligned}&\mathit{any} \;:\; \{A : \mathsf{Set}\} \to \mathcal{P}\,A\\ &\mathit{any} \;=\; \mathit{const}\ \top\end{aligned}$$

The subset $\mathit{any} : \mathcal{P}\,A$ associates the unit type $\top$ with every element of A. Since $\top$ is inhabited, *any* can produce any element of A. While $\mathcal{P}$ cannot be made into a conventional monad [Moggi, 1991; Wadler, 1992] because it is not an endofunctor, it can still be equipped with the usual monadic programming combinators, giving rise to a **relative monad** [Altenkirch et al., 2010]:

- The monadic unit is defined as

  $$\begin{aligned}&\mathit{return} \;:\; \{A : \mathsf{Set}\} \to A \to \mathcal{P}\,A\\ &\mathit{return} \;=\; \_\!\equiv\!\_\end{aligned}$$

  The subset $\mathit{return}\ a : \mathcal{P}\,A$ for some $a : A$ simplifies to $\lambda\, a' \mapsto a \equiv a'$, so a is the only member of the subset.

- The monadic bind is defined as

  $$\_\!\gg\!=\!\_ \;:\; \{A\ B : \mathsf{Set}\} \to \mathcal{P}\,A \to (A \to \mathcal{P}\,B) \to \mathcal{P}\,B$$

$$\_{\gg\!=}\_\ \{A\}\ s\,f \;=\; \lambda\, b \mapsto \Sigma[\, a : A\,]\ s\ a \times f\ a\ b$$

If $s : \mathcal{P}\,A$ and $f : A \to \mathcal{P}\,B$, then the subset $s \gg\!= f : \mathcal{P}\,B$ is the disjoint union of all the subsets $f\ a : \mathcal{P}\,B$ where a ranges over the elements of A that belong to s; that is, an element $b : B$ is a member of $s \gg\!= f$ exactly when there exists some $a : A$ belonging to s such that b is a member of $f\ a$.

It is easy to show that the two combinators satisfy the (relative) monad laws up to pointwise isomorphism, whose proofs we omit from the presentation. On top of *return* and $\_{\gg\!=}\_$, the functorial map of $\mathcal{P}$ is defined as

$$\begin{aligned}&\_\langle\$\rangle\_ \;:\; \{A\ B : \mathsf{Set}\} \to (A \to B) \to \mathcal{P}\,A \to \mathcal{P}\,B\\ &f\ \langle\$\rangle\ s \;=\; s \gg\!= \lambda\, a \mapsto \mathit{return}\ (f\ a)\end{aligned}$$

and we also define a two-argument version for convenience:

$$\begin{aligned}&\_\langle\$\rangle^2\_ \;:\; \{A\ B\ C : \mathsf{Set}\} \to (A \to B \to C) \to \mathcal{P}\,A \to \mathcal{P}\,B \to \mathcal{P}\,C\\ &f\ \langle\$\rangle^2\ s\ t \;=\; s \gg\!= \lambda\, a \mapsto t \gg\!= \lambda\, b \mapsto \mathit{return}\ (f\ a\ b)\end{aligned}$$

The notation is a reference to applicative functors [McBride and Paterson, 2008], allowing us to think of functorial maps of $\mathcal{P}$ as applications of pure functions to effectful arguments.

We will mainly use families of relations between families of sets:

$$\begin{aligned}&\_\rightsquigarrow\_ \;:\; \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}_1\\ &X \rightsquigarrow Y \;=\; \forall\ \{i\} \to X\ i \to \mathcal{P}\,(Y\ i)\end{aligned}$$

which is the usual generalisation of $\_\rightrightarrows\_$ to allow nondeterminacy. Here we define several relational operators that we will need. relations vs families of relations

- Since functions are deterministic relations, we have the following combinator *fun* that lifts functions to relations using *return*.

$$\begin{aligned}&\mathit{fun} \;:\; \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightrightarrows Y) \to (X \rightsquigarrow Y)\\ &\mathit{fun}\ f\ x \;=\; \mathit{return}\ (f\ x)\end{aligned}$$

- The identity relation is just the identity function lifted by *fun*.

$$\begin{aligned}&\mathit{idR} \;:\; \{I : \mathsf{Set}\}\ \{X : I \to \mathsf{Set}\} \to (X \rightsquigarrow X)\\ &\mathit{idR} \;=\; \mathit{fun}\ \mathit{id}\end{aligned}$$

$$
\begin{aligned}
&\mathit{mapR} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to \\
&\qquad\qquad (X \rightsquigarrow Y) \to [\![\ D\ ]\!]\ X \to \mathscr{P}([\![\ D\ ]\!]\ Y) \\
&\mathit{mapR}\ (\mathsf{v}\ [])\qquad\quad R\ \blacksquare \qquad\quad = \mathit{return}\ \blacksquare \\
&\mathit{mapR}\ (\mathsf{v}\ (i :: is))\ R\ (x\ ,\ xs)\ =\ \_,\_\ \langle\$\rangle^2\ (R\ x)\ (\mathit{mapR}\ (\mathsf{v}\ is)\ R\ xs) \\
&\mathit{mapR}\ (\sigma\ S\ D)\qquad R\ (s\ ,\ xs)\ =\ (\_,\_\ s)\ \langle\$\rangle\ (\mathit{mapR}\ (D\ s)\ R\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbb{R} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightsquigarrow Y) \to (\mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y) \\
&\mathbb{R}\ D\ R\ \{i\}\ =\ \mathit{mapR}\ (D\ i)\ R
\end{aligned}
$$

Figure 5.1 Definition for relators.

- Composition of relations is easily defined with $\_{\gg\!=}\_$: computing $R \bullet S$ on input x is first computing $S\ x$ and then feeding the result to R.

  $$
  \begin{aligned}
  &\_\bullet\_ : \{I : \mathsf{Set}\}\ \{X\ Y\ Z : I \to \mathsf{Set}\} \to (Y \rightsquigarrow Z) \to (X \rightsquigarrow Y) \to (X \rightsquigarrow Z) \\
  &(R \bullet S)\ x\ =\ S\ x \gg\!= R
  \end{aligned}
  $$

- Some relations do not carry obvious computational meaning, which we can still define pointwise, like the meet of two relations:

  $$
  \begin{aligned}
  &\_\cap\_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightsquigarrow Y) \to (X \rightsquigarrow Y) \to (X \rightsquigarrow Y) \\
  &(R \cap S)\ x\ y\ =\ R\ x\ y \times S\ x\ y
  \end{aligned}
  $$

- Unlike a function, which distinguishes between input and output, inherently a relation treats its domain and codomain symmetrically. This is reflected by the presence of the following **converse** operator:

  $$
  \begin{aligned}
  &\_^{\circ} : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightsquigarrow Y) \to (Y \rightsquigarrow X) \\
  &(R^{\circ})\ y\ x\ =\ R\ x\ y
  \end{aligned}
  $$

  A relation can thus be "run backwards" simply by taking its converse. The nondeterministic and bidirectional nature of relations makes them a powerful and concise language for specifications, as will be demonstrated in Sections 5.3.2 and 5.3.3.

- We will also need **relators**, i.e., functorial maps on relations:

  $$\mathbb{R} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to$$

$$(X \rightsquigarrow Y) \to (\mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y)$$

If $R : X \rightsquigarrow Y$, the relation $\mathbb{R}\ D\ R : \mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y$ applies R to the recursive positions of its input, leaving everything else intact. The definition of $\mathbb{R}$ is shown in Figure 5.1. For example, if $D = \mathit{ListD}\ A$, then $\mathbb{R}\ (\mathit{ListD}\ A)$ is, up to isomorphism,

$$\begin{aligned}
&\mathbb{R}\ (\mathit{ListD}\ A) : \{X\ Y : I \to \mathsf{Set}\} \to \\
&\qquad\qquad (X \rightsquigarrow Y) \to (\mathbb{F}\ (\mathit{ListD}\ A)\ X \rightsquigarrow \mathbb{F}\ (\mathit{ListD}\ A)\ Y) \\
&\mathbb{R}\ (\mathit{ListD}\ A)\ R\ (\text{`}\mathsf{nil}\ ,\ \blacksquare) = \mathit{return}\ (\text{`}\mathsf{nil}\ ,\ \blacksquare) \\
&\mathbb{R}\ (\mathit{ListD}\ A)\ R\ (\text{`}\mathsf{cons}\ ,\ a\ ,\ x\ ,\ \blacksquare) = (\lambda\, y \mapsto \text{`}\mathsf{cons}\ ,\ a\ ,\ y\ ,\ \blacksquare)\ \langle\$\rangle\ (R\ x)
\end{aligned}$$

Laws and theorems about relational programs are formulated with relational inclusion:

$$\begin{aligned}
&_\subseteq_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ (R\ S : X \rightsquigarrow Y) \to \mathsf{Set} \\
&R \subseteq S = \forall\ \{i\} \to (x : X\ i)\ (y : Y\ i) \to R\ x\ y \to S\ x\ y
\end{aligned}$$

or equivalence of relations, i.e., two-way inclusion:

$$\begin{aligned}
&_\simeq_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ (R\ S : X \rightsquigarrow Y) \to \mathsf{Set} \\
&R \simeq S = (R \subseteq S) \times (S \subseteq R)
\end{aligned}$$

With relational inclusion, many concepts can be expressed in a surprisingly concise way. For example, a relation R is a preorder if it is reflexive and transitive. In relational terms, these two conditions are expressed simply as

$$\mathit{idR} \subseteq R \qquad \mathsf{and} \qquad R \bullet R \subseteq R$$

and are easily manipulable in calculations. Another important notion is **monotonic algebras** [Bird and de Moor, 1997, Section 7.2]: an algebra $S : \mathbb{F}\ D\ X \rightsquigarrow X$ is **monotonic** on $R : X \rightsquigarrow X$ (usually an ordering) if

$$S \bullet \mathbb{R}\ D\ R \subseteq R \bullet S$$

which says that if two input values to S have their recursive positions related by R and are otherwise equal, then the output values would still be related by R. In the context of optimisation problems, monotonicity can be used to capture the **principle of optimality**, as will be shown in Section 5.3.3.

probably a simple example of relational calculation here?

mutual

$$
\begin{aligned}
&(\!|\_|\!) : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightsquigarrow X) \to (\mu\ D \rightsquigarrow X)\\
&(\!|\_|\!)\ \{I\}\ \{D\}\ R\ \{i\}\ (\mathsf{con}\ ds) = \mathit{mapFoldR}\ D\ (D\ i)\ R\ ds \gg\!= R\\
&\mathit{mapFoldR} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ (D' : \mathsf{RDesc}\ I) \to\\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightsquigarrow X) \to [\![\ D'\ ]\!]\ (\mu\ D) \to \mathscr{P}\,([\![\ D'\ ]\!]\ X)\\
&\mathit{mapFoldR}\ D\ (\mathsf{v}\ [])\qquad R\ \blacksquare \qquad = \mathit{return}\ \blacksquare\\
&\mathit{mapFoldR}\ D\ (\mathsf{v}\ (i :: is))\ R\ (d\ ,\ ds) = \_,\_\ \langle\$\rangle^2\ ((\!|\,R\,|\!)\ d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{mapFoldR}\ D\ (\mathsf{v}\ is)\ f\ ds)\\
&\mathit{mapFoldR}\ D\ (\sigma\ S\ D')\quad R\ (s\ ,\ ds) = (\_,\_\ s)\ \langle\$\rangle\ (\mathit{mapFoldR}\ D\ (D'\ s)\ f\ ds)
\end{aligned}
$$

Figure 5.2 Definition of relational folds.

Having defined relations as nondeterministic mappings, it is straightforward to rewrite the datatype-generic *fold* with the subset combinators to obtain a relational version, which is denoted by "banana brackets" [Meijer et al., 1991]:

$$(\!|\_|\!) : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightsquigarrow X) \to (\mu\ D \rightsquigarrow X)$$

The definition of $(\!|\_|\!)$ is shown in Figure 5.2. For example, the relational fold on lists is, up to isomorphism,

$$
\begin{aligned}
&(\!|\_|\!)\ \{\top\}\ \{\mathit{ListD}\ A\} : \{X : \top \to \mathsf{Set}\} \to\\
&\qquad\qquad (\mathbb{F}\ (\mathit{ListD}\ A)\ X \rightsquigarrow X) \to (\mu\ (\mathit{ListD}\ A) \rightsquigarrow X)\\
&(\!|\,R\,|\!)\ [] \qquad = R\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare)\\
&(\!|\,R\,|\!)\ (a :: as) = (\!|\,R\,|\!)\ as \gg\!= \lambda\, x \mapsto R\ (\text{‘}\mathsf{cons}\ ,\ a\ ,\ x\ ,\ \blacksquare)
\end{aligned}
$$

The functional and relational fold operators are related by the following lemma:

$$
\begin{aligned}
&\mathit{fun\text{-}preserves\text{-}fold} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X : I \to \mathsf{Set}\} \to\\
&\qquad (f : \mathbb{F}\ D\ X \rightrightarrows X)\ \{i : I\}\ (d : \mu\ D\ i)\ (x : X\ i) \to\\
&\qquad \mathit{fun}\ (\mathit{fold}\ f)\ d\ x \cong (\!|\,\mathit{fun}\ f\,|\!)\ d\ x
\end{aligned}
$$

which is a strengthened version of $\mathit{fun}\ (\mathit{fold}\ f) \simeq (\!|\,\mathit{fun}\ f\,|\!)$.

5.2 Definition of relational algebraic ornaments

We now turn to relational algebraic ornamentation, the key construct that bridges internalist and relational programming. Let

$$R : \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ X) \rightsquigarrow \mathit{const}\ X$$

(where X : Set) be a relational algebra for lists. We can define a datatype of "algebraic lists" as

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{AlgList}\ A\ R : X \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{AlgList}\ A\ R\ x \ni \mathsf{nil}\ (\mathit{rnil} : R\ (\text{'}\mathsf{nil}\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\quad \mid\ \mathsf{cons}\ (a : A)\ (x' : X) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{rcons} : R\ (\text{'}\mathsf{cons}\ ,\ a\ ,\ x'\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{as} : \mathsf{AlgList}\ A\ R\ x')
\end{array}$$

There is an ornament from lists to algebraic lists which marks the fields *rnil*, x', and *rcons* in AlgList as additional and refines the index of the recursive position from $\blacksquare$ to x'. The optimised predicate (Section 3.3.1) for this ornament is

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{AlgListP}\ A\ R : X \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{AlgListP}\ A\ R\ x\ [\,] \qquad\quad\ \ni \mathsf{nil}\ (\mathit{rnil} : R\ (\text{'}\mathsf{nil}\ ,\ \blacksquare)\ x) \\
\quad \mathsf{AlgListP}\ A\ R\ x\ (a :: \mathit{as}) \ni \mathsf{cons}\ (x' : X) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{rcons} : R\ (\text{'}\mathsf{cons}\ ,\ a\ ,\ x'\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (p : \mathsf{AlgListP}\ A\ R\ x'\ \mathit{as})
\end{array}$$

A simple argument by induction shows that AlgListP $A\ R\ x\ \mathit{as}$ is in fact isomorphic to $(\!|R|\!)\ \mathit{as}\ x$ for any as : List A and $x : X$. By predicate swapping for the refinement semantics of the ornament from lists to algebraic lists (Section 3.3), we get

$$\mathsf{AlgList}\ A\ R\ x \cong \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ (\!|R|\!)\ \mathit{as}\ x \tag{5.1}$$

for all $x : X$. That is, an algebraic list is exactly a plain list and a proof that the list folds to x using the algebra R. The traditional bottom-up vector datatype is a special case of AlgList — define

$$\begin{array}{l}
\mathit{length\text{-}alg} : \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat} \\
\mathit{length\text{-}alg}\ (\text{'}\mathsf{nil}\quad ,\qquad\quad \blacksquare) = \mathsf{zero}
\end{array}$$

$$
\begin{array}{l}
\mathit{algROD} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I)\ \{J : I \to \mathsf{Set}\} \to \\
\qquad\qquad (\llbracket\ D\ \rrbracket\ J \to \mathsf{Set}) \to \mathsf{ROrnDesc}\ (\Sigma\ I\ J)\ \mathsf{outl}\ D \\
\mathit{algROD}\ (\mathsf{v}\ \mathit{is})\quad \{J\}\ P\ =\ \Delta[\mathit{js} : \mathbb{P}\ \mathit{is}\ J]\ \Delta[\_ : P\ \mathit{js}] \\
\qquad\qquad\qquad\qquad\qquad \mathsf{v}\ (\mathbb{P}\text{-}\mathit{map}\ (\lambda\ \{i\}\ j \mapsto \mathsf{ok}\ (i\ ,j))\ \mathit{is}\ \mathit{js}) \\
\mathit{algROD}\ (\sigma\ S\ D)\qquad P\ =\ \sigma[s : S]\ \mathit{algROD}\ (D\ s)\ (\mathit{curry}\ P\ s) \\
\mathit{algOD} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{J : I \to \mathsf{Set}\} \to \\
\qquad\qquad (\mathbb{F}\ D\ J \rightsquigarrow J) \to \mathsf{OrnDesc}\ (\Sigma\ I\ J)\ \mathsf{outl}\ D \\
\mathit{algOD}\ D\ R\ (\mathsf{ok}\ (i\ ,j))\ =\ \mathit{algROD}\ (D\ i)\ (\lambda\ \mathit{js} \mapsto R\ \mathit{js}\ j)
\end{array}
$$

Figure 5.3 Definitions for algebraic ornamentation.

$$\mathit{length\text{-}alg}\ (\text{‘}\mathsf{cons}\ ,a\ ,n\ ,\blacksquare)\ =\ \mathsf{suc}\ n$$

and then $\mathsf{AlgList}\ A\ (\mathit{fun}\ \mathit{length\text{-}alg})$ is exactly $\mathsf{Vec}'\ A$. By (5.1) we have the isomorphisms

$$\mathsf{Vec}'\ A\ n\ \cong\ \Sigma[\mathit{as} : \mathsf{List}\ A]\ (\!|\ \mathit{fun}\ \mathit{length\text{-}alg}\ |\!)\ \mathit{as}\ n$$

for all $n : \mathsf{Nat}$, from which we can derive

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

by *fun-preserves-fold*, where $\mathit{length}\ =\ \mathit{fold}\ \mathit{length\text{-}alg}$.

The above can be generalised to all datatypes encoded by the Desc universe. Let $D : \mathsf{Desc}\ I$ be a description and $R : \mathbb{F}\ D\ X \rightsquigarrow X$ (where $X : I \to \mathsf{Set}$) an algebra. The **algebraic ornamentation** of D with R is an ornamental description

$$\mathit{algOD}\ D\ R : \mathsf{OrnDesc}\ (\Sigma\ I\ X)\ \mathsf{outl}\ D$$

(where $\mathsf{outl} : \Sigma\ I\ X \to I$). The optimised predicate for $\lceil\ \mathit{algOD}\ D\ R\ \rceil$ is pointwise isomorphic to $(\!|\ R\ |\!)$, i.e.,

$$\mathsf{OptP}\ \lceil \mathit{algOD}\ D\ R \rceil\ (\mathsf{ok}\ (i\ ,x))\ d\ \cong\ (\!|\ R\ |\!)\ d\ x$$

for all $i : I$, $x : X\ i$, and $d : \mu\ D\ i$. These isomorphisms give rise to a family of predicate swaps for the refinement semantics of $\lceil \mathit{algOD}\ D\ R \rceil$, so we arrive at the following conversion isomorphisms

$$\mu\ \lfloor \mathit{algOD}\ D\ R \rfloor\ (i\ ,x)\ \cong\ \Sigma[d : \mu\ D\ i]\ (\!|\ R\ |\!)\ d\ x \tag{5.2}$$

for all $i : I$ and $x : X\ i$. The definition of *algOD*, shown in Figure 5.3, is an adaptation and generalisation of McBride's original definition of functional algebraic ornaments [2011]. Roughly speaking, it retains all the fields of the base description and inserts before every v

- a new field of indices for the recursive positions (e.g., the field x' in AlgList) and
- another new field requesting a proof that
  - the indices supplied in the previous field and
  - the values for the fields originally in the base description

  computes to the targeted index through R (e.g., the fields *rnil* and *rcons* in AlgList).

summary and some gluing to the next section

5.3 Examples

5.3.1 The Fold Fusion Theorem

Revise using upgrades.

As a first example of bridging internalist programming with relational calculation through algebraic ornamentation, let us consider the **Fold Fusion Theorem** [Bird and de Moor, 1997, Section 6.2]: Let D : Desc I be a description, $R : X \rightsquigarrow Y$ a relation, and $S : \mathbb{F}\ D\ X \rightsquigarrow X$ and $T : \mathbb{F}\ D\ Y \rightsquigarrow Y$ algebras. If R is a homomorphism from S to T, i.e.,

$$R \cdot S \simeq T \cdot \mathbb{R}\ D\ R$$

which is referred to as the **fusion condition**, then we have

$$R \cdot (\!|\, S \,|\!) \simeq (\!|\, T \,|\!)$$

The above is, in fact, a corollary of two variations of Fold Fusion that replace relational equivalence in the statement of the theorem with relational inclusion. One of the variations is

$$R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R \quad \rightarrow \quad R \bullet (\!|\, S \,|\!) \subseteq (\!|\, T \,|\!)$$

This can be used with (5.2) to derive a conversion between algebraically ornamented datatypes:

$$\begin{array}{l} \mathit{algOD\text{-}fusion\text{-}}{\subseteq}\; D\; R\; S\; T \;: \\ \quad R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R \rightarrow \\ \quad \{i : I\}\; (x : X\, i) \rightarrow \mu\; \lfloor \mathit{algOD}\; D\; S \rfloor\; (i\,,x) \rightarrow \\ \quad (y : Y\, i) \rightarrow R\; x\; y \rightarrow \mu\; \lfloor \mathit{algOD}\; D\; T \rfloor\; (i\,,y) \end{array}$$

The other variation of Fold Fusion simply reverses the direction of inclusion:

$$R \bullet S \supseteq T \bullet \mathbb{R}\, D\, R \quad \rightarrow \quad R \bullet (\!|\, S \,|\!) \supseteq (\!|\, T \,|\!)$$

which translates to the conversion

$$\begin{array}{l} \mathit{algOD\text{-}fusion\text{-}}{\supseteq}\; D\; R\; S\; T \;: \\ \quad R \bullet S \supseteq T \bullet \mathbb{R}\, D\, R \rightarrow \\ \quad \{i : I\}\; (y : Y\, i) \rightarrow \mu\; \lfloor \mathit{algOD}\; D\; T \rfloor\; (i\,,y) \rightarrow \\ \quad \Sigma[\, x : X\, i \,]\; \mu\; \lfloor \mathit{algOD}\; D\; S \rfloor\; (i\,,x) \times R\; x\; y \end{array}$$

For a simple example, suppose that we need a "bounded" vector datatype, i.e., lists indexed with an upper bound on their length. A quick thought might lead to this definition

$$\begin{array}{l} \mathsf{BVec} : \mathsf{Set} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set} \\ \mathsf{BVec}\; A\; m \;=\; \mu\; \lfloor \mathit{algOD}\; (\mathit{ListD}\; A)\; (\mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg}) \rfloor\; (\blacksquare\,, m) \end{array}$$

where $\mathit{geq} = \lambda\, x\, y \rightarrow x \leqslant y : \mathit{const}\; \mathsf{Nat} \rightsquigarrow \mathit{const}\; \mathsf{Nat}$ maps a natural number x to any natural number that is at least x. The isomorphisms (5.2) specialise for BVec to

$$\mathsf{BVec}\; A\; m \cong \Sigma[\, \mathit{as} : \mathsf{List}\; A \,]\; (\!|\, \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \,|\!)\; \mathit{as}\; m$$

for all m : Nat. But is BVec really the bounded vectors? Indeed it is, because we can deduce

$$\mathit{geq} \bullet (\!|\, \mathit{fun}\; \mathit{length\text{-}alg} \,|\!) \simeq (\!|\, \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \,|\!)$$

by Fold Fusion. The fusion condition is

$$\mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \simeq \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \bullet \mathbb{R}\; (\mathit{ListD}\; A)\; \mathit{geq}$$

relator laws and various monotonicity need to be stated earlier

The left-to-right inclusion is easily calculated as follows:

$$
\begin{array}{ll}
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \\
\subseteq & \{\ \mathit{idR}\ \text{identity}\ \} \\
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \cdot \mathit{idR} \\
\subseteq & \{\ \text{relator preserves identity}\ \} \\
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \cdot \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{idR} \\
\subseteq & \{\ \mathit{geq}\ \text{reflexive}\ \} \\
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \cdot \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{geq}
\end{array}
$$

And from right to left:

$$
\begin{array}{ll}
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \cdot \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{geq} \\
\subseteq & \{\ \mathit{fun}\ \mathit{length\text{-}alg}\ \text{monotonic on}\ \mathit{geq}\ \} \\
 & \mathit{geq} \cdot \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg} \\
\subseteq & \{\ \mathit{geq}\ \text{transitive}\ \} \\
 & \mathit{geq} \cdot \mathit{fun}\ \mathit{length\text{-}alg}
\end{array}
$$

Note that these calculations are good illustrations of the power of relational calculation despite their simplicity — they are straightforward symbolic manipulations, hiding details like quantifier reasoning behind the scenes. As demonstrated by the AoPA library [Mu et al., 2009], they can be faithfully formalised with preorder reasoning combinators in Agda and used to discharge the fusion conditions of *algOD-fusion-*$\subseteq$ and *algOD-fusion-*$\supseteq$. Hence we get two conversions, one of type

$$\mathsf{Vec}\ A\ n \to (n \leqslant m) \to \mathsf{BVec}\ A\ m$$

which relaxes a vector of length n to a bounded vector whose length is bounded above by some m that is at least n, and the other of type

$$\mathsf{BVec}\ A\ m \to \Sigma[\, n : \mathsf{Nat}\,]\ \mathsf{Vec}\ A\ n \times (n \leqslant m)$$

which converts a bounded vector whose length is at most m to a vector of length precisely n and guarantees that n is at most m.

Just constraint transformation; base data do not change

5.3.2 The Streaming Theorem for list metamorphisms

A **metamorphism** [Gibbons, 2007] is an unfold after a fold — it consumes a data structure to compute an intermediate value and then produces a new data structure using the intermediate value as the seed. In this section we will restrict ourselves to metamorphisms consuming and producing lists. As Gibbons noted, (list) metamorphisms in general cannot be automatically optimised in terms of time and space, but under certain conditions it is possible to refine a list metamorphism to a **streaming algorithm** — which can produce an initial segment of the output list without consuming all of the input list — or a parallel algorithm [Nakano, 2013]. In the rest of this section, we prove the **Streaming Theorem** [Bird and Gibbons, 2003, Theorem 30] by implementing the streaming algorithm given by the theorem with algebraic ornamented lists such that the algorithm satisfies its metamorphic specification by construction.

Our first step is to formulate a metamorphism as a relational specification of the streaming algorithm.

- The fold part needs a twist since using the conventional fold — known as the **right fold** for lists since the direction of computation on a list is from right to left (cf. wind direction) — does not easily give rise to a streaming algorithm. This is because we wish to talk about "partial consumption" naturally: for a list, partial consumption means examining and removing some elements of the list to get a sub-list on which we can resume consumption, and the natural way to do this is to consume the list from the left, examining and removing head elements and keeping the tail. We should thus use the **left fold** instead, which is usually defined as

  $$\begin{array}{lcl} \mathit{foldl} \;:\; \{A\;X \;:\; \mathsf{Set}\} \to (X \to A \to X) \to X \to \mathsf{List}\;A \to X \\ \mathit{foldl}\,f\;x\;[\,] & = & x \\ \mathit{foldl}\,f\;x\;(a :: as) & = & \mathit{foldl}\,f\;(f\;x\;a)\;as \end{array}$$

  The connection to the conventional fold (and thus algebraic ornamentation) is not lost, however — it is well known that a left fold can be alternatively implemented as a right fold by turning a list into a chain of functions of type $X \to X$ transforming the initial value to the final result:

$$
\begin{aligned}
&\mathit{foldl\text{-}alg} : \{A\ X : \mathsf{Set}\} \to (X \to A \to X) \to \\
&\qquad\qquad \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ (X \to X)) \rightrightarrows \mathit{const}\ (X \to X) \\
&\mathit{foldl\text{-}alg}\ f\ (\text{`}\mathsf{nil}\ \ ,\qquad\quad \blacksquare) = \mathit{id} \\
&\mathit{foldl\text{-}alg}\ f\ (\text{`}\mathsf{cons}\ , a\ , h\ , \blacksquare) = h \circ \mathit{flip}\ f\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl} : \{A\ X : \mathsf{Set}\} \to (X \to A \to X) \to X \to \mathsf{List}\ A \to X \\
&\mathit{foldl}\ f\ x\ \mathit{as} = \mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as}\ x
\end{aligned}
$$

The left fold can thus be linked to the relational fold by

$$\mathit{fun}\ (\mathit{foldl}\ f\ x) \simeq \mathit{fun}\ (\lambda\, h \mapsto h\ x) \bullet (\!|\, \mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)\, |\!) \tag{5.3}$$

- The unfold part is approximated by the converse of a relational fold: given a list coalgebra $g : \mathit{const}\ X \rightrightarrows \mathbb{F}\ (\mathit{ListD}\ B)\ (\mathit{const}\ X)$ for some $X : \mathsf{Set}$, we take its converse, turning it into a relational algebra, and use the converse of the relational fold with this algebra.

  $$(\!|\, \mathit{fun}\ g^{\circ}\, |\!)^{\circ} : \mathit{const}\ X \rightsquigarrow \mathit{const}\ (\mathsf{List}\ A)$$

  This is only an approximation because, while the relation does produce a list, the resulting list is inductive rather than coinductive, so the relation is actually a **well-founded** unfold, which is incapable of producing an infinite list.

Thus, given a "left algebra" for consuming $\mathsf{List}\ A$

$$f : X \to A \to X$$

and a coalgebra for producing $\mathsf{List}\ B$

$$g : \mathit{const}\ X \rightrightarrows \mathbb{F}\ (\mathit{ListD}\ B)\ (\mathit{const}\ X)$$

which together satisfy a **streaming condition** that we will see later, the streaming algorithm we implement, which takes as input the initial value $x : X$ for the left fold, should be included in the following metamorphic relation:

$$\mathit{meta}\ f\ g\ x = (\!|\, \mathit{fun}\ g^{\circ}\, |\!)^{\circ} \bullet \mathit{fun}\ (\mathit{foldl}\ f\ x) : \mathit{const}\ (\mathsf{List}\ A) \rightsquigarrow \mathit{const}\ (\mathsf{List}\ B)$$

Next we devise a type for the streaming algorithm that fully guarantees its correctness. By (5.3), the specification $\mathit{meta}\ f\ g\ x$ is equivalent to

$$(\!|\, \mathit{fun}\ g^{\circ}\, |\!)^{\circ} \bullet \mathit{fun}\ (\lambda\, h \mapsto h\ x) \bullet (\!|\, \mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)\, |\!)$$

Inspecting the above relation, we see that a conforming program takes a $\mathsf{List}\ A$ that folds to some $h : X \to X$ with $\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)$ and computes a $\mathsf{List}\ B$ that folds to $h\ x : X$ with $\mathit{fun}\ g^{\circ}$. We are thus going to implement the streaming algorithm as

$$\begin{aligned}&\mathit{stream}\ f\ g : (x : X)\ \{h : X \to X\} \to\\ &\qquad\qquad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (h\ x)\end{aligned}$$

from which we can extract

$$\mathit{stream}'\ f\ g : X \to \mathsf{List}\ A \to \mathsf{List}\ B$$

which is guaranteed to satisfy

$$\mathit{fun}\ (\mathit{stream}'\ f\ g\ x) \subseteq \mathit{meta}\ f\ g\ x$$

The extraction of $\mathit{stream}'\ f\ g$ from $\mathit{stream}\ f\ g$ is done with the help of the conversion isomorphisms (5.2) for the two algebraic list datatypes involved:

$$\begin{aligned}&\mathit{consumption\text{-}iso} :\\ &\quad (h : X \to X) \to\\ &\quad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \cong \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ \mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as} \equiv h\\ &\mathit{production\text{-}iso} :\\ &\quad (x : X) \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ x \cong \Sigma[\,\mathit{bs} : \mathsf{List}\ B\,]\ (\!|\mathit{fun}\ g^{\circ}|\!)\ \mathit{bs}\ x\end{aligned}$$

(where *consumption-iso* has been simplified by *fun-preserves-fold*). Given $x : X$, what $\mathit{stream}'\ f\ g\ x$ does is

- lifting the input $\mathit{as} : \mathsf{List}\ A$ to an algebraic list of type

  $$\mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ (\mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as})$$

  using the right-to-left direction of $\mathit{consumption\text{-}iso}\ (\mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as})$ (with the equality proof obligation discharged trivially by refl),
- transforming this algebraic list to a new one of type

  $$\mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (\mathit{foldl}\ f\ x\ \mathit{as})$$

  using $\mathit{stream}\ f\ g\ x$, and
- demoting the new algebraic list to $\mathsf{List}\ B$ using the left-to-right direction of $\mathit{production\text{-}iso}\ (\mathit{foldl}\ f\ x\ \mathit{as})$.

The use of *production-iso* in the last step ensures that the result $\mathit{stream}'\ f\ g\ x\ \mathit{as}\ :\ \mathsf{List}\ B$ satisfies

$$(\!|\, \mathit{fun}\ g^{\circ} \,|\!)\ (\mathit{stream}'\ f\ g\ x\ \mathit{as})\ (\mathit{foldl}\ f\ x\ \mathit{as})$$

which easily implies

$$((\!|\, \mathit{fun}\ g^{\circ} \,|\!)^{\circ} \bullet \mathit{fun}\ (\mathit{foldl}\ f\ x))\ \mathit{as}\ (\mathit{stream}'\ f\ g\ x\ \mathit{as})$$

i.e., $\mathit{fun}\ (\mathit{stream}'\ f\ g\ x)\ \subseteq\ \mathit{meta}\ f\ g\ x$, as required.

What is left is the implementation of $\mathit{stream}\ f\ g$. Operationally, we maintain a state of type X (and hence requires an initial state as an input to the function), and we can try either

- to update the state by consuming elements of A with f, or
- to produce elements of B (and transit to a new state) by applying g to the state.

Since we want $\mathit{stream}\ f\ g$ to be as productive as possible, we should always try to produce elements of B with g first, and only try to consume elements of A with f when g produces nothing. In Agda:

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathit{stream}\ f\ g\ :\ (x\ :\ X)\ \{h\ :\ X \to X\} \to} \\
\multicolumn{6}{l}{\qquad\qquad\quad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (h\ x)} \\
\mathit{stream}\ f\ g\ x & \mathit{as} & \mathbf{with}\ g\ x & \mid\ \mathit{inspect}\ g\ x & & \\
\mathit{stream}\ f\ g\ x\ \{h\}\ \mathit{as} & & \mid\ \mathsf{next}\ b\ x' & \mid\ [\ \mathit{gxeq}\ ] & = & \mathsf{cons}\ b\ (h\ x')\ \{\ \}_0 \\
 & & & & & \quad (\mathit{stream}\ f\ g\ x'\ \mathit{as}) \\
\mathit{stream}\ f\ g\ x & (\mathsf{nil}\quad \mathsf{refl}\quad) & \mid\ \mathsf{nothing} & \mid\ [\ \mathit{gxeq}\ ] & = & \mathsf{nil}\ \mathit{gxeq} \\
\mathit{stream}\ f\ g\ x & (\mathsf{cons}\ a\ h'\ \mathsf{refl}\ \mathit{as}) & \mid\ \mathsf{nothing} & \mid\ [\ \mathit{gxeq}\ ] & = & \mathit{stream}\ f\ g\ (f\ x\ a)\ \mathit{as}
\end{array}
$$

We match $g\ x$ with either of the two patterns $\mathsf{next}\ b\ x'\ =\ (\text{'}\mathsf{cons}\ ,\ b\ ,\ x'\ ,\ \blacksquare)$ and $\mathsf{nothing}\ =\ (\text{'}\mathsf{nil}\ ,\ \blacksquare)$.

Agda doesn't really allow this, though.

- If the result is $\mathsf{next}\ b\ x'$, we should emit b and and use x' as the new state; the recursively computed algebraic list is indexed with $h\ x'$, and we are left with a proof obligation of type $g\ (h\ x)\ \equiv\ \mathsf{next}\ b\ (h\ x')$ at Goal 0; we will come back to this proof obligation later.
- If the result is $\mathsf{nothing}$, we should attempt to consume the input list.

$$\begin{tikzcd}[column sep=huge, row sep=huge] x \arrow[r, mapsto, "\text{produce } b \text{ with } g"] \arrow[d, mapsto, "\text{consume } as \text{ with } h"'] & x' \arrow[d, mapsto, "\text{consume } as \text{ with } h"] \\ h\ x \arrow[r, mapsto, "\text{produce } b \text{ with } g"'] & h\ x' \end{tikzcd}$$

Figure 5.4 State transitions involved in commutativity of production and consumption (cf. Gibbons [2007, Figures 1 and 2]).

- If the input list is empty, implying that the index h of its type is just id, both production and consumption have ended, so we return an empty list. The nil constructor requires a proof of $(fun\ g\ ^{\circ})\ \mathsf{nothing}\ (h\ x)$, which reduces to $g\ x\ \equiv\ \mathsf{nothing}$ and is discharged with the help of the "inspect idiom" in Agda's standard library (which, in a **with**-matching, gives a proof that the term being matched (in this case $g\ x$) is propositionally equal to the matched pattern (in this case $\mathsf{nothing}$)).
- Otherwise the input list is nonempty, implying that h is $h' \circ flip\ f\ a$ where a is the head of the input list, and we should continue with the new state $f\ x\ a$, keeping the tail for further consumption. Typing directly works out because the index of the recursive result $h'\ (f\ x\ a)$ and the required index $(h' \circ flip\ f\ a)\ x$ are definitionally equal.

Now we look at Goal 0. We have

$$gxeq\ :\ g\ x\ \equiv\ \mathsf{next}\ b\ x'$$

in the context, and need to prove

$$g\ (h\ x)\ \equiv\ \mathsf{next}\ b\ (h\ x')$$

This is commutativity of production and consumption (see Figure 5.4): The function $h\ :\ X \rightarrow X$ is the state transformation resulting from consumption of the input list as. From the initial state x, we can either

- apply g to x to **produce** b and reach a new state x', and then apply h to **consume** the list and update the state to $h\ x'$, or

- apply h to **consume** the list and update the state to $h\ x$, and then apply g to $h\ x$ to **produce** an element and reach a new state,

and we need to prove that the outcomes are the same: doing production using g and consumption using h in whichever order should emit the same element and reach the same final state. This cannot be true in general, so we should impose some commutativity condition on f and g, which is called the **streaming condition**:

$$
\begin{aligned}
&\mathit{StreamingCondition}\ f\ g\ :\ \mathsf{Set}\\
&\mathit{StreamingCondition}\ f\ g\ =\\
&\quad (a\ :\ A)\ (b\ :\ B)\ (x\ x'\ :\ X) \to g\ x \equiv \mathsf{next}\ b\ x' \to g\ (f\ x\ a) \equiv \mathsf{next}\ b\ (f\ x'\ a)
\end{aligned}
$$

The streaming condition is commutativity of one step of production and consumption, whereas the proof obligation at Goal 0 is commutativity of one step of production and multiple steps of consumption (of the entire list), so we perform a straightforward induction to extend the streaming condition along the axis of consumption:

$$
\begin{aligned}
&\mathit{streaming\text{-}lemma}\ :\\
&\quad (b\ :\ B)\ (x\ x'\ :\ X) \to g\ x \equiv \mathsf{next}\ b\ x' \to\\
&\quad \{h\ :\ X \to X\} \to \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \to g\ (h\ x) \equiv \mathsf{next}\ b\ (h\ x')\\
&\mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{nil}\qquad \mathsf{refl}\quad)\ =\ \mathit{eq}\\
&\mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{cons}\ a\ h\ \mathsf{refl}\ \mathit{as})\ =\\
&\quad \mathit{streaming\text{-}lemma}\ b\ (f\ x\ a)\ (f\ x'\ a)\ (\mathit{streaming\text{-}condition}\ f\ g\ a\ b\ x\ x'\ \mathit{eq})\ \mathit{as}
\end{aligned}
$$

where $\mathit{streaming\text{-}condition}\ :\ \mathit{StreamingCondition}\ f\ g$ is a proof term that should be supplied along with f and g in the beginning. Goal 0 is then discharged by the term $\mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{gxeq}\ \mathit{as}$.

We have thus completed the implementation of the Streaming Theorem, except that $\mathit{stream}\ f\ g$ is non-terminating, as there is no guarantee that g produces only a finite number of elements. In our setting, where the output list is specified to be finite, we can additionally require that g is well-founded and revise *stream* accordingly (see, e.g., Nordström [1988]); the general way out is to switch to coinductive datatypes to allow the output list to be infinite, which, however, falls outside the scope of this thesis.

It is interesting to compare our implementation with the proofs of Bird and Gibbons [2003]. While their Lemma 29 turns explicitly into our *streaming-lemma*, their Theorem 30 goes implicitly into the typing of *stream* and no longer needs special attention. The structure of *stream* already matches that of Bird and Gibbons's proof of their Theorem 30, and the principled type design using algebraic ornamentation elegantly loads the proof onto the structure of *stream* — this is internalism at its best.

5.3.3 The minimum coin change problem

Suppose that we have an unlimited number of 1-penny, 2-pence, and 5-pence coins, modelled by the following datatype:

data Coin : Set **where** 1p 2p 5p : Coin

Given n : Nat, the **minimum coin change problem** asks for the least number of coins that make up n pence. We can give a relational specification of the problem with the following minimisation operator:

$$min\_ \bullet \Lambda\_ \ : \ \{I : \mathsf{Set}\} \ \{X\ Y : I \to \mathsf{Set}\} \ (R : Y \rightsquigarrow Y) \ (S : X \rightsquigarrow Y) \to (X \rightsquigarrow Y)$$
$$(min\ R \bullet \Lambda\ S)\ x\ y \ = \ S\ x\ y \times (\forall\ y' \to S\ x\ y' \to R\ y'\ y)$$

An input $x : X\ i$ for some $i : I$ is mapped by $min\ R \bullet \Lambda\ S$ to $y : Y\ i$ if y is a possible result in $S\ x : \mathcal{P}(Y\ i)$ and is the smallest such result under R, in the sense that any y' in $S\ x : \mathcal{P}(Y\ i)$ must satisfy $R\ y'\ y$. (We think of R as mapping larger inputs to smaller outputs.) Intuitively, we can think of $min\ R \bullet \Lambda\ S$ as consisting of two steps: the first step $\Lambda\ S$ computes the set of all possible results yielded by S, and the second step $min\ R$ nondeterministically chooses a minimum result from that set. We use bags of coins as the type of solutions, and represent them as decreasingly ordered lists indexed with an upper bound. (This is a deliberate choice to make the derivation work, but one would naturally turn to this representation having attempted to apply the **Greedy Theorem**, which will be introduced shortly.) If we define the ordering on coins as

$$\_\leqslant_{\mathsf{C}}\_ \ : \ \mathsf{Coin} \rightarrow \mathsf{Coin} \rightarrow \mathsf{Set}$$
$$c \leqslant_{\mathsf{C}} d \ = \ \mathit{value}\ c \leqslant \mathit{value}\ d$$

where the values of the coins are defined by

$$\begin{aligned}
&\mathit{value} \ : \ \mathsf{Coin} \rightarrow \mathsf{Nat}\\
&\mathit{value}\ \mathsf{1p} \ = \ 1\\
&\mathit{value}\ \mathsf{2p} \ = \ 2\\
&\mathit{value}\ \mathsf{5p} \ = \ 5
\end{aligned}$$

then the datatype of coin bags we use is

$$\begin{aligned}
&\mathit{CoinBagOD} \ : \ \mathsf{OrnDesc}\ \mathsf{Coin}\ !\ (\mathit{ListD}\ \mathsf{Coin})\\
&\mathit{CoinBagOD} \ = \ \mathit{OrdListOD}\ \mathsf{Coin}\ (\mathit{flip}\ _\leqslant_{\mathsf{C}}_)
\end{aligned}$$

$$\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{CoinBag} \ : \ \mathsf{Coin} \rightarrow \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{CoinBag}\ c \ \ni \ \mathsf{nil}\\
&\qquad\qquad\qquad \mid \ \mathsf{cons}\ (d \ : \ \mathsf{Coin})\ (\mathit{leq} \ : \ d \leqslant_{\mathsf{C}} c)\ (b \ : \ \mathsf{CoinBag}\ d)
\end{aligned}$$

The base functor for CoinBag is

$$\begin{aligned}
&\mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor \ : \ (\mathsf{Coin} \rightarrow \mathsf{Set}) \rightarrow (\mathsf{Coin} \rightarrow \mathsf{Set})\\
&\mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ X\ c \ =\\
&\quad \Sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \ \ \mapsto \ \top\\
&\qquad\qquad\qquad ;\ \text{‘}\mathsf{cons} \ \mapsto \ \Sigma[\,d : \mathsf{Coin}\,]\ (d \leqslant_{\mathsf{C}} c) \times X\ d \times \top\ \}
\end{aligned}$$

The total value of a coin bag is the sum of the values of the coins in the bag, which is computed by a (functional) fold:

$$\begin{aligned}
&\mathit{total\text{-}value\text{-}alg} \ : \ \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{nil} \ , \qquad\qquad\quad \blacksquare) \ = \ 0\\
&\mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{cons}\ , d\ , _\ , n\ , \blacksquare) \ = \ \mathit{value}\ d + n
\end{aligned}$$

$$\begin{aligned}
&\mathit{total\text{-}value} \ : \ \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{total\text{-}value} \ = \ \mathit{fold}\ \mathit{total\text{-}value\text{-}alg}
\end{aligned}$$

and the number of coins in a coin bag is also computed by a fold:

$$\begin{aligned}
&\mathit{size\text{-}alg} \ : \ \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{size\text{-}alg}\ (\text{‘}\mathsf{nil} \ , \qquad\qquad\quad \blacksquare) \ = \ 0\\
&\mathit{size\text{-}alg}\ (\text{‘}\mathsf{cons}\ , _\ , _\ , n\ , \blacksquare) \ = \ 1 + n
\end{aligned}$$

$$
\begin{aligned}
&\mathit{size} \;:\; \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat} \\
&\mathit{size} \;=\; \mathit{fold}\ \mathit{size\text{-}alg}
\end{aligned}
$$

The specification of the minimum coin change problem can now be written as

$$
\begin{aligned}
&\mathit{min\text{-}coin\text{-}change} \;:\; \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathsf{CoinBag} \\
&\mathit{min\text{-}coin\text{-}change} \;=\; \mathit{min}\ (\mathit{fun}\ \mathit{size}^{\circ} \bullet \mathit{leq} \bullet \mathit{fun}\ \mathit{size}) \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value}^{\circ})
\end{aligned}
$$

where $\mathit{leq} = \mathit{geq}^{\circ} : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$ maps a natural number n to any natural number that is at most n. Intuitively, given an input $n : \mathsf{Nat}$, the relation $\mathit{fun}\ \mathit{total\text{-}value}^{\circ}$ computes an arbitrary coin bag whose total value is n, so *min-coin-change* first computes the set of all such coin bags and then chooses from the set a coin bag whose size is smallest. Our goal, then, is to write a functional program $f : \mathit{const}\ \mathsf{Nat} \rightrightarrows \mathsf{CoinBag}$ such that $\mathit{fun}\ f \subseteq \mathit{min\text{-}coin\text{-}change}$, and then $f\ \{\mathsf{5p}\} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p}$ would be a solution. (The type $\mathsf{CoinBag}\ \mathsf{5p}$ contains all coin bags, since $\mathsf{5p}$ is the largest denomination and hence a trivial upper bound on the content of bags.) Of course, we may guess what f should look like, but its correctness proof is much harder. Can we construct the program and its correctness proof in a more manageable way?

The plan

In traditional relational program derivation, we would attempt to refine the specification *min-coin-change* to some simpler relational program and then to an executable functional program by applying algebraic laws and theorems. With algebraic ornamentation, however, there is a new possibility: if we can derive that, for some algebra $R : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathit{const}\ \mathsf{Nat}$,

$$
(\!|\, R \,|\!)^{\circ} \subseteq \mathit{min\text{-}coin\text{-}change} \tag{5.4}
$$

then we can manufacture a new datatype

$$
\begin{aligned}
&\mathit{GreedyBagOD} \;:\; \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathsf{outl}\ \lfloor \mathit{CoinBagOD} \rfloor \\
&\mathit{GreedyBagOD} \;=\; \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{GreedyBag} \;:\; \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{GreedyBag}\ c\ n \;=\; \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,\ n)
\end{aligned}
$$

and construct a function of type

$$\mathit{greedy} : (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedyBag}\ c\ n$$

from which we can assemble a solution

$$\begin{aligned} &\mathit{sol} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p} \\ &\mathit{sol} = \mathit{forget}\ \lceil \mathit{GreedyBagOD} \rceil \circ \mathit{greedy}\ \mathsf{5p} \end{aligned}$$

The program *sol* satisfies the specification because of the following argument: For any $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$, by (5.2) we have

$$\mathsf{GreedyBag}\ c\ n \ \cong\ \Sigma[\, b : \mathsf{CoinBag}\ c\,]\ \ (\!| R |\!)\ b\ n$$

In particular, since the first half of the left-to-right direction of the isomorphism is $\mathit{forget}\ \lceil \mathit{GreedyBagOD} \rceil$, we have

$$(\!| R |\!)\ (\mathit{forget}\ \lceil \mathit{GreedyBagOD} \rceil\ g)\ n$$

for any $g : \mathsf{GreedyBag}\ c\ n$. Substituting g by $\mathit{greedy}\ \mathsf{5p}\ n$, we get

$$(\!| R |\!)\ (\mathit{sol}\ n)\ n$$

which implies, by (5.4),

$$\mathit{min\text{-}coin\text{-}change}\ n\ (\mathit{sol}\ n)$$

i.e., *sol* satisfies the specification. Thus all we need to do to solve the minimum coin change problem is

- refine the specification *min-coin-change* to the converse of a fold, i.e., find the algebra R in (5.4), and
- construct the internalist program *greedy*.

Refining the specification

The key to refining *min-coin-change* to the converse of a fold lies in the following version of the **Greedy Theorem**, which is essentially a specialisation of Bird and de Moor's Theorem 10.1 [1997]: Let $D : \mathsf{Desc}\ I$ be a description, $R : \mu\ D \rightsquigarrow \mu\ D$ a preorder, and $S : \mathbb{F}\ D\ X \rightsquigarrow X$ an algebra. Consider the specification

$$min\ R \bullet \Lambda\ (\llparenthesis\, S\, \rrparenthesis\ ^\circ)$$

That is, given an input value $x : X\ i$ for some $i : I$, we choose a minimum under R among all those elements of $\mu\ D\ i$ that computes to x through $\llparenthesis\, S\, \rrparenthesis$. The Greedy Theorem states that, if the initial algebra

$$\alpha\ =\ fun\ \mathsf{con} : \mathbb{F}\ D\ (\mu\ D) \rightsquigarrow \mu\ D$$

is monotonic on R, i.e.,

$$\alpha \bullet \mathbb{R}\ D\ R\ \subseteq\ R \bullet \alpha$$

and there is a relation (ordering) $Q : \mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ X$ such that the **greedy condition**

$$\alpha \bullet \mathbb{R}\ D\ (\llparenthesis\, S\, \rrparenthesis\ ^\circ) \bullet (Q \cap (S^\circ \bullet S))^\circ\ \subseteq\ R^\circ \bullet \alpha \bullet \mathbb{R}\ D\ (\llparenthesis\, S\, \rrparenthesis\ ^\circ)$$

is satisfied, then we have

$$\llparenthesis\ (min\ Q \bullet \Lambda\ (S^\circ))^\circ\ \rrparenthesis^\circ\ \subseteq\ min\ R \bullet \Lambda\ (\llparenthesis\, S\, \rrparenthesis\ ^\circ)$$

Here we offer an intuitive explanation of the Greedy Theorem, but the theorem admits an elegant calculational proof, which can be faithfully reprised in Agda. The monotonicity condition states that if $ds : \mathbb{F}\ D\ (\mu\ D)\ i$ for some $i : I$ is better than $ds' : \mathbb{F}\ D\ (\mu\ D)\ i$ under $\mathbb{R}\ D\ R$, i.e., ds and ds' are equal except that the recursive positions of ds are all better than the corresponding recursive positions of ds' under R, then $\mathsf{con}\ ds : \mu\ D\ i$ would be better than $\mathsf{con}\ ds' : \mu\ D\ i$ under R. This implies that, when solving the optimisation problem, better solutions to subproblems would lead to a better solution to the original problem, so the **principle of optimality** applies — to reach an optimal solution, it suffices to find optimal solutions to subproblems, and we are entitled to use the converse of a fold to find optimal solutions recursively. The greedy condition further states that there is an ordering Q on the ways of decomposing the problem which has significant influence on the quality of solutions: Suppose that there are two decompositions xs and $xs' : \mathbb{F}\ D\ X\ i$ of some problem $x : X\ i$ for some $i : I$, i.e., both xs and xs' are in $S^\circ\ x : \mathcal{P}\,(\mathbb{F}\ D\ X\ i)$, and assume that xs is better than xs' under Q. Then for any solution resulting from xs' (computed by $\alpha \bullet \mathbb{R}\ D\ (\llparenthesis\, S\, \rrparenthesis\ ^\circ)$) there always exists a better solution resulting from xs, so ignoring xs' would only rule out worse solutions. The greedy condition thus

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{CoinBag'View} : \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\}\ \{l : \mathsf{Nat}\} \to \mathsf{CoinBag'}\ c\ n\ l \to \mathsf{Set}\ \textbf{where}} \\
\quad \mathsf{empty} & : \{c : \mathsf{Coin}\} \to \mathsf{CoinBag'View}\ \{c\}\ \{0\}\ \{0\}\ \mathsf{bnil} & \\
\quad \mathsf{1p1p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{1p} \leqslant_{\mathsf{C}} \mathsf{1p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{1p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{1p}\}\ \{1+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{1p}\ lep\ b) & \\
\quad \mathsf{1p2p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{1p} \leqslant_{\mathsf{C}} \mathsf{2p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{1p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{2p}\}\ \{1+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{1p}\ lep\ b) & \\
\quad \mathsf{2p2p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{2p} \leqslant_{\mathsf{C}} \mathsf{2p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{2p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{2p}\}\ \{2+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{2p}\ lep\ b) & \\
\quad \mathsf{1p5p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{1p} \leqslant_{\mathsf{C}} \mathsf{5p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{1p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{5p}\}\ \{1+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{1p}\ lep\ b) & \\
\quad \mathsf{2p5p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{2p} \leqslant_{\mathsf{C}} \mathsf{5p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{2p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{5p}\}\ \{2+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{2p}\ lep\ b) & \\
\quad \mathsf{5p5p} & : \{m\ l : \mathsf{Nat}\}\ \{lep : \mathsf{5p} \leqslant_{\mathsf{C}} \mathsf{5p}\} & \\
 & \quad (b : \mathsf{CoinBag'}\ \mathsf{5p}\ m\ l) \to \mathsf{CoinBag'View}\ \{\mathsf{5p}\}\ \{5+m\}\ \{1+l\}\ (\mathsf{bcons}\ \mathsf{5p}\ lep\ b) &
\end{array}
$$

Figure 5.5 The view datatype on CoinBag′.

guarantees that we will arrive at an optimal solution by always choosing the best decomposition, which is done by $min\ Q \bullet \Lambda\ (S^{\circ}) : X \rightsquigarrow \mathbb{F}\ D\ X$.

Back to the minimum coin change problem: By *fun-preserves-fold*, the specification *min-coin-change* is equivalent to

$$min\ (fun\ size^{\circ} \bullet leq \bullet fun\ size) \bullet \Lambda\ ((\!|\ fun\ total\text{-}value\text{-}alg\ |\!)^{\circ})$$

which matches the form of the generic specification given in the Greedy Theorem, so we try to discharge the two conditions of the theorem. The monotonicity condition reduces to monotonicity of *fun size-alg* on *leq*, and can be easily proved either by relational calculation or pointwise reasoning. As for the greedy condition, an obvious choice for Q is an ordering that leads us to choose the largest possible denomination, so we go for

$$
\begin{array}{l}
Q : \mathbb{F}\ \lfloor CoinBagOD \rfloor\ (const\ \mathsf{Nat}) \rightsquigarrow \mathbb{F}\ \lfloor CoinBagOD \rfloor\ (const\ \mathsf{Nat}) \\
Q\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) = return\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) \\
Q\ (\text{‘}\mathsf{cons}\ ,\ d\ ,\ \_) = (\lambda\ e\ rest \mapsto \text{‘}\mathsf{cons}\ ,\ e\ ,\ rest)\ \langle\$\rangle^{2}\ (\_\leqslant_{\mathsf{C}}\_\ d)\ any
\end{array}
$$

```
greedy-lemma : (c d : Coin) → c ≤C d → (m n : Nat) → value c + m ≡ value d + n →
               (l : Nat) (b : CoinBag′ c m l) → Σ[ l′ : Nat ] CoinBag′ d n l′ × (l′ ≤ l)
greedy-lemma c   d   c≤d m        n       eq   l          b with view-ordered-coin c d c≤d
greedy-lemma .1p .1p _   .n       n       refl l          b_(CoinBag′ 1p n l) | 1p1p = { Σ[ l′ : Nat ] CoinBag′ 1p n l′ × (l′ ≤ l) }0
greedy-lemma .1p .2p _   .(1 + n) n       refl l          b | 1p2p with view-CoinBag′ b
greedy-lemma .1p .2p _   .(1 + n) n       refl .(1 + l″) ._ | 1p2p | 1p1p {.n} {l″} b_(CoinBag′ 1p n l″) =
                                                          { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ≤ 1 + l″) }1
greedy-lemma .1p .5p _   .(4 + n) n       refl l          b | 1p5p with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n       refl ._         ._ | 1p5p | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n       refl ._         ._ | 1p5p | 1p1p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n       refl ._         ._ | 1p5p | 1p1p ._ | 1p1p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n       refl .(4 + l″) ._ | 1p5p | 1p1p ._ | 1p1p ._ | 1p1p ._ | 1p1p {.n} {l″} b_(CoinBag′ 1p n l″) =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 4 + l″) }2
greedy-lemma .2p .2p _   .n       n       refl l          b_(CoinBag′ 2p n l) | 2p2p = { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ≤ l) }3
greedy-lemma .2p .5p _   .(3 + n) n       refl l          b | 2p5p with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n       refl ._         ._ | 2p5p | 1p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n       refl ._         ._ | 2p5p | 1p2p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n       refl .(3 + l″) ._ | 2p5p | 1p2p ._ | 1p1p ._ | 1p1p {.n} {l″} b_(CoinBag′ 1p n l″) =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 3 + l″) }4
greedy-lemma .2p .5p _   .(3 + n) n       refl ._         ._ | 2p5p | 2p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n       refl .(2 + l″) ._ | 2p5p | 2p2p ._ | 1p2p {.n} {l″} b_(CoinBag′ 2p n l″) =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 2 + l″) }5
greedy-lemma .2p .5p _   .(4 + k) .(1 + k) refl .(2 + l″) ._ | 2p5p | 2p2p ._ | 2p2p {k} {l″} b_(CoinBag′ 2p k l″) =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p (1 + k) l′ × (l′ ≤ 2 + l″) }6
greedy-lemma .5p .5p _   .n       n       refl l          b_(CoinBag′ 5p n l) | 5p5p = { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ l) }7
```

Figure 5.6 Cases of *greedy-lemma*, generated semi-automatically by Agda's interactive case-split mechanism. Goal types are shown in the interaction points, and the types of some pattern variables are shown in subscript beside them.

where, in the cons case, the output is required to be also a cons node, and the coin at its head position must be one that is no smaller than the coin d at the head position of the input. It is non-trivial to prove the greedy condition by relational calculation. Here we offer instead a brute-force yet conveniently expressed case analysis by dependent pattern matching. Define a new datatype $\mathsf{CoinBag}'$ by composing two algebraic ornaments on $\lfloor \mathit{CoinBagOD} \rfloor$ in parallel:

$$
\begin{aligned}
&\mathit{CoinBag'OD} \;:\; \mathsf{OrnDesc}\;(\mathsf{outl} \bowtie \mathsf{outl})\;\mathit{pull}\;\lfloor \mathit{CoinBagOD} \rfloor \\
&\mathit{CoinBag'OD} \;=\; \lceil \mathit{algOD}\;\lfloor \mathit{CoinBagOD} \rfloor\;(\mathit{fun}\;\mathit{total\text{-}value\text{-}alg}) \rceil \otimes \\
&\qquad\qquad\qquad\; \lceil \mathit{algOD}\;\lfloor \mathit{CoinBagOD} \rfloor\;(\mathit{fun}\;\mathit{size\text{-}alg}) \rceil
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{CoinBag}' \;:\; \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{CoinBag}' \;=\; \mu\;\lfloor \mathit{CoinBag'OD} \rfloor\;(\mathsf{ok}\;(c\;,\;n)\;,\;\mathsf{ok}\;(c\;,\;l))
\end{aligned}
$$

whose two constructors can be specialised to

$$
\begin{aligned}
&\mathsf{bnil} \;\;\;:\; \{c \,:\, \mathsf{Coin}\} \to \mathsf{CoinBag}'\;c\;0\;0 \\
&\mathsf{bcons} \;:\; \{c \,:\, \mathsf{Coin}\}\;\{n\;l \,:\, \mathsf{Nat}\} \to (d \,:\, \mathsf{Coin}) \to d \leqslant_{\mathsf{C}} c \to \\
&\qquad\quad \mathsf{CoinBag}'\;d\;n\;l \to \mathsf{CoinBag}'\;c\;(\mathit{value}\;d + n)\;(1 + l)
\end{aligned}
$$

By predicate swapping using the modularity isomorphisms (Section 3.3.2) and *fun-preserves-fold*, $\mathsf{CoinBag}'$ is characterised by the isomorphisms

$$
\mathsf{CoinBag}'\;c\;n\;l \;\cong\; \Sigma[\,b : \mathsf{CoinBag}\;c\,]\;\;(\mathit{total\text{-}value}\;b \;\equiv\; n) \times (\mathit{size}\;b \;\equiv\; l) \tag{5.5}
$$

for all $c \,:\, \mathsf{Coin}$, $n \,:\, \mathsf{Nat}$, and $l \,:\, \mathsf{Nat}$. Hence a coin bag of type $\mathsf{CoinBag}'\;c\;n\;l$ contains l coins that are no larger than c and sum up to n pence. The greedy condition then essentially reduces to this lemma:

$$
\begin{aligned}
&\mathit{greedy\text{-}lemma} \;:\; (c\;d \,:\, \mathsf{Coin}) \to c \leqslant_{\mathsf{C}} d \to \\
&\qquad\qquad\qquad\quad (m\;n \,:\, \mathsf{Nat}) \to \mathit{value}\;c + m \;\equiv\; \mathit{value}\;d + n \to \\
&\qquad\qquad\qquad\quad (l \,:\, \mathsf{Nat})\;(b \,:\, \mathsf{CoinBag}'\;c\;m\;l) \to \\
&\qquad\qquad\qquad\quad \Sigma[\,l' : \mathsf{Nat}\,]\;\;\mathsf{CoinBag}'\;d\;n\;l' \times (l' \leqslant l)
\end{aligned}
$$

That is, given a problem (i.e., a value to be represented by coins), if $c \,:\, \mathsf{Coin}$ is a choice of decomposition (i.e., the first coin used) no better than $d \,:\, \mathsf{Coin}$ (i.e., $c \leqslant_{\mathsf{C}} d$ — recall that we prefer larger denominations), and $b \,:\, \mathsf{CoinBag}'\;c\;m\;l$ is a solution of size l to the remaining subproblem m resulting from choosing c, then there is a solution to the remaining subproblem n resulting from choosing d whose size l' is no greater than l. We define two **views** [McBride and McKinna, 2004] — or "customised pattern matching" — to aid the analysis:

- The first view analyses a proof of $c \leqslant_{\mathsf{C}} d$ and exhausts all possibilities of c and d,

  $$\begin{array}{l} \mathbf{data}\ \mathsf{CoinOrderedView}\ :\ \mathsf{Coin} \rightarrow \mathsf{Coin} \rightarrow \mathsf{Set}\ \mathbf{where} \\ \quad \mathsf{1p1p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{1p}\ \mathsf{1p} \\ \quad \mathsf{1p2p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{1p}\ \mathsf{2p} \\ \quad \mathsf{1p5p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{1p}\ \mathsf{5p} \\ \quad \mathsf{2p2p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{2p}\ \mathsf{2p} \\ \quad \mathsf{2p5p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{2p}\ \mathsf{5p} \\ \quad \mathsf{5p5p}\ :\ \mathsf{CoinOrderedView}\ \mathsf{5p}\ \mathsf{5p} \end{array}$$

  $$\textit{view-ordered-coin}\ :\ (c\ d\ :\ \mathsf{Coin}) \rightarrow c \leqslant_{\mathsf{C}} d \rightarrow \mathsf{CoinOrderedView}\ c\ d$$

  where the covering function *view-ordered-coin* is written by standard pattern matching on c and d.

- The second view analyses some $b\ :\ \mathsf{CoinBag}'\ c\ n\ l$ and exhausts all possibilities of c, n, l, and the first coin in b (if any). The view datatype $\mathsf{CoinBag'View}$ is shown in Figure 5.5, and the covering function

  $$\begin{array}{l} \textit{view-CoinBag}'\ : \\ \quad \{c\ :\ \mathsf{Coin}\}\ \{n\ l\ :\ \mathsf{Nat}\}\ (b\ :\ \mathsf{CoinBag}'\ c\ n\ l) \rightarrow \mathsf{CoinBag'View}\ b \end{array}$$

  is again written by standard pattern matching.

Given these two views, the function *greedy-lemma* can be split into eight cases by first exhausting all possibilities of c and d with *view-ordered-coin* and then analysing the content of b with *view-CoinBag*$'$. Figure 5.6 shows the case-split tree generated semi-automatically by Agda; the detail is explained as follows:

- At Goal 0 (and similarly Goals 3 and 7), the input bag is $b\ :\ \mathsf{CoinBag}'\ \mathsf{1p}\ n\ l$, and we should produce a $\mathsf{CoinBag}'\ \mathsf{1p}\ n\ l'$ for some $l'\ :\ \mathsf{Nat}$ such that $l' \leqslant l$. This is easy because b itself is a suitable bag.
- At Goal 1 (and similarly Goals 2, 4, and 5), the input bag has type $\mathsf{CoinBag}'\ \mathsf{1p}\ (1 + n)\ l$, i.e., the coins in the bag are no larger than $\mathsf{1p}$ and the total value is $1 + n$. The bag must contain $\mathsf{1p}$ as its first coin; let the rest

of the bag be b : $\mathsf{CoinBag}'\ \mathsf{1p}\ n\ l''$. At this point Agda can deduce that l must be $1 + l''$. Now we can return b as the result after the upper bound on its coins is relaxed from 1p to 2p, which is done by

$$\mathit{relax} : \{c\ d : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \rightarrow c \leqslant_{\mathsf{C}} d \rightarrow \mathsf{CoinBag}'\ c\ n\ l \rightarrow \mathsf{CoinBag}'\ d\ n\ l$$

- The remaining Goal 6 is the most interesting one: The input bag has type $\mathsf{CoinBag}'\ \mathsf{2p}\ (3 + n)\ l$, which in this case contains two 2-pence coins, and the rest of the bag is b : $\mathsf{CoinBag}'\ \mathsf{2p}\ k\ l''$. Agda deduces that n must be $1 + k$ and l must be $2 + l''$. We thus need to add a penny to b to increase its total value to $1 + k$, which is done by

  $$\begin{aligned}&\mathit{add\text{-}penny} :\\ &\quad \{c : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \rightarrow \mathsf{CoinBag}'\ c\ n\ l \rightarrow \mathsf{CoinBag}'\ c\ (1 + n)\ (1 + l)\end{aligned}$$

  and relax the bound of *add-penny b* from 2p to 5p.

The above case analysis may look tedious, but note that Agda is able to

- produce all the cases (modulo some cosmetic revisions) after the programmer decides to use the two views and instructs Agda to do case splitting accordingly, and
- manage all the bookkeeping and deductions about the total value and the size of bags with dependent pattern matching,

so the overhead on the programmer's side is actually less than it seems. The greedy condition can now be discharged by pointwise reasoning, using (5.5) to interface with *greedy-lemma*. We conclude that the Greedy Theorem is applicable, and obtain

$$(\!\lbrack\, (\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}^{\circ}))^{\circ} \,\rbrack\!)^{\circ} \subseteq \mathit{min\text{-}coin\text{-}change}$$

We have thus found the algebra

$$R = (\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}^{\circ}))^{\circ}$$

which will help us to construct the final internalist program.

Constructing the internalist program

As planned, we synthesise a new datatype by ornamenting $\mathsf{CoinBag}$ using the algebra R derived above:

$$
\begin{aligned}
&\mathit{GreedyBagOD} \ : \ \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathsf{outl}\ \lfloor \mathit{CoinBagOD} \rfloor \\
&\mathit{GreedyBagOD} \ = \ \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{GreedyBag} \ : \ \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{GreedyBag}\ c\ n \ = \ \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,\ n)
\end{aligned}
$$

whose two constructors can be given the following types:

$$
\begin{aligned}
\mathsf{gnil} \ :\ & \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\} \to \\
& \mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{nil}\ ,\ \blacksquare) \equiv n \to \\
& ((\mathit{ns} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
& \quad \mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv n \to Q\ \mathit{ns}\ (\text{`}\mathsf{nil}\ ,\ \blacksquare)) \to \\
& \mathsf{GreedyBag}\ c\ n \\
\mathsf{gcons} \ :\ & \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\}\ (d : \mathsf{Coin})\ (d{\leqslant}c : d \leqslant_{\mathsf{C}} c) \to \\
& \{n' : \mathsf{Nat}\} \to \mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{cons}\ ,\ d\ ,\ d{\leqslant}c\ ,\ n') \equiv n \to \\
& ((\mathit{ns} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
& \quad \mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv n \to Q\ \mathit{ns}\ (\text{`}\mathsf{cons}\ ,\ d\ ,\ d{\leqslant}c\ ,\ n')) \to \\
& \mathsf{GreedyBag}\ d\ n' \to \mathsf{GreedyBag}\ c\ n
\end{aligned}
$$

and implement the greedy algorithm by

$$
\mathit{greedy} \ : \ (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedyBag}\ c\ n
$$

Let us first simplify the two constructors of $\mathsf{GreedyBag}$. Each of the two constructors has two additional proof obligations coming from the algebra R:

- For gnil,
  - the first obligation $\mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{nil}\ ,\ \blacksquare) \equiv n$ reduces to $0 \equiv n$, so we may discharge the obligation by specialising n to 0;
  - for the second obligation, ns is necessarily $(\text{`}\mathsf{nil}\ ,\ \blacksquare)$ if $\mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv 0$, and indeed $Q$ maps $(\text{`}\mathsf{nil}\ ,\ \blacksquare)$ to $(\text{`}\mathsf{nil}\ ,\ \blacksquare)$, so the second obligation can be discharged as well.

  We thus obtain a simplified version of gnil:

$$\mathsf{gnil}' : \{c : \mathsf{Coin}\} \to \mathsf{GreedyBag}\ c\ 0$$

- For gcons,
  - the first obligation reduces to $\mathit{value}\ d + n' \equiv n$, so we may just specialise n to $\mathit{value}\ d + n'$ and discharge the obligation;
  - for the second obligation, any ns satisfying $\mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv \mathit{value}\ d + n'$ must be of the form $(\text{‘}\mathsf{cons}\ , e\ , e{\leqslant}c\ , m'\ , \blacksquare)$ for some $e : \mathsf{Coin}$, $e{\leqslant}c : e \leqslant_{\mathsf{C}} c$, and $m' : \mathsf{Nat}$ since the right-hand side $\mathit{value}\ d + n'$ of the equality is non-zero, and Q maps ns to $(\text{‘}\mathsf{cons}\ , d\ , d{\leqslant}c\ , n'\ , \blacksquare)$ if $e \leqslant_{\mathsf{C}} d$, so d should be the largest "usable" coin if this obligation is to be discharged. We say that $d : \mathsf{Coin}$ is **usable** with respect to some $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$ if d is bounded above by c and can be part of a solution to the problem for n pence:

    $$\mathit{UsableCoin} : \mathsf{Nat} \to \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set}$$
    $$\mathit{UsableCoin}\ n\ c\ d = (d \leqslant_{\mathsf{C}} c) \times (\Sigma[\, n' : \mathsf{Nat}\,]\ \mathit{value}\ d + n' \equiv n)$$

    The obligation can then be rewritten as

    $$(e : \mathsf{Coin}) \to \mathit{UsableCoin}\ (\mathit{value}\ d + n')\ c\ e \to e \leqslant_{\mathsf{C}} d$$

    which requires that d is the largest usable coin with respect to c and $\mathit{value}\ d + n'$. This obligation is the only one that cannot be trivially discharged, since it requires computation of the largest usable coin.

  We thus specialise gcons to

$$\begin{aligned}\mathsf{gcons}' : {} & \{c : \mathsf{Coin}\}\ (d : \mathsf{Coin}) \to d \leqslant_{\mathsf{C}} c \to \\ & \{n' : \mathsf{Nat}\} \to \\ & ((e : \mathsf{Coin}) \to \mathit{UsableCoin}\ (\mathit{value}\ d + n')\ c\ e \to e \leqslant_{\mathsf{C}} d) \to \\ & \mathsf{GreedyBag}\ d\ n' \to \mathsf{GreedyBag}\ c\ (\mathit{value}\ d + n')\end{aligned}$$

Because of gcons′, we are directed to implement a function *maximum-coin* that computes the largest usable coin with respect to any $c : \mathsf{Coin}$ and non-zero $n : \mathsf{Nat}$:

$$\begin{aligned}& \mathit{maximum\text{-}coin} : \\ & \quad (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to n > 0 \to \\ & \quad \Sigma[\, d : \mathsf{Coin}\,]\ \mathit{UsableCoin}\ n\ c\ d \times ((e : \mathsf{Coin}) \to \mathit{UsableCoin}\ n\ c\ e \to e \leqslant_{\mathsf{C}} d)\end{aligned}$$

This takes some theorem proving but is overall a typical Agda exercise in dealing with natural numbers and ordering. Finally, the greedy algorithm is implemented as the following internalist program, which repeatedly uses *maximum-coin* to find the largest usable coin and unfolds a GreedyBag:

```
greedy : (c : Coin) (n : Nat) → GreedyBag c n
greedy c n = <-rec P f n c
  where
    P : Nat → Set
    P n = (c : Coin) → GreedyBag c n
    f : (n : Nat) → ((n′ : Nat) → n′ < n → P n′) → P n
    f n               rec c with compare-with-zero n
    f .0              rec c | is-zero = gnil′
    f n               rec c | above-zero n>z with maximum-coin c n n>z
    f .(value d + n′) rec c | above-zero n>z | d , (d⩽c , n′ , refl) , guc =
                                     gcons′ d d⩽c guc (rec n′ { }8 d)
```

In *greedy*, the combinator

```
<-rec : (P : Nat → Set) →
        ((n : Nat) → ((n′ : Nat) → n′ < n → P n′) → P n) →
        (n : Nat) → P n
```

is for well-founded recursion on $\_<\_$, and the function

```
compare-with-zero : (n : Nat) → ZeroView n
```

is a covering function for the view

```
data ZeroView : Nat → Set where
  is-zero    : ZeroView 0
  above-zero : {n : Nat} → n > 0 → ZeroView n
```

At Goal 8, Agda deduces that n is $\mathit{value}\ d + n'$ and demands that we prove $n' < \mathit{value}\ d + n'$ in order to make the recursive call, which is easily discharged since $\mathit{value}\ d > 0$.

5.4 Discussion

compare the McBride [2011] version (compatible with the two-constructor universe) and the Dagand and McBride [2012b] version of algebraic ornamentation in terms of "quality" (amount of σ's used); proof-relevant Algebra of Programming (e.g., *fun-preserves-fold*; linking to the next chapter); related work: Atkey et al. [2012]

Chapter 6

Categorical equivalence of ornaments and relational algebras

Consider the AlgList datatype in Section 5.2 again. The way it is refined relative to the plain list datatype looks canonical, in the sense that any variation of the list datatype can be programmed as a special case of AlgList: we can choose whatever index set we want by setting the carrier of the algebra R; and by carefully programming R, we can insert fields into the list datatype that add more information or put restriction on fields and indices. For example, if we want some new information in the nil case, we can program R such that $R\ (\mathit{nil} - \mathit{tag}\ ,\ \blacksquare)\ x$ contains a field requesting that information; if, in the cons case, we need the targeted index x, the head element a, and the index x' of the recursive position to be related in some way, we can program R such that $R\ (\mathit{cons} - \mathit{tag}\ ,\ a\ ,\ x')\ x$ expresses that relationship.

The above observation leads to the following general theorem: Let $O : \mathsf{Orn}\ e\ D\ E$ be an ornament from $D : \mathsf{Desc}\ I$ to $E : \mathsf{Desc}\ J$. There is a **classifying algebra** for O

$$\mathit{clsAlg}\ O : \mathbb{F}\ D\ (\mathsf{InvImage}\ e) \rightsquigarrow \mathsf{InvImage}\ e$$

such that there are isomorphisms

$$\mu\ \lfloor \mathit{algOrn}\ D\ (\mathit{clsAlg}\ O) \rfloor\ (e\ j\ ,\ \mathsf{ok}\ j)\ \cong\ \mu\ E\ j$$

for all $j : J$. That is, the algebraic ornamentation of D using the classifying algebra derived from O produces a datatype isomorphic to $\mu\ E$, so intuitively the algebraic ornament has the same content as O. We may interpret this theorem as saying that algebraic ornaments are "complete" for the ornament language: any relationship between datatypes that can be described by an ornament can be described up to isomorphism by an algebraic ornament.

The completeness theorem brings up a nice algebraic intuition about inductive families. Consider the ornament from lists to vectors, for example. This ornament specifies that the type $\mathsf{List}\ A$ is refined by the collection of types $\mathsf{Vec}\ A\ n$ for all $n : \mathsf{Nat}$. A list, say $a :: b :: [\,] : \mathsf{List}\ A$, can be reconstructed as a vector by starting in the type $\mathsf{Vec}\ A\ \mathsf{zero}$ as $[\,]$, jumping to the next type $\mathsf{Vec}\ A\ (\mathsf{suc}\ \mathsf{zero})$ as $b :: [\,]$, and finally landing in $\mathsf{Vec}\ A\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}))$ as $a :: b :: [\,]$. The list is thus **classified** as having length 2, as computed by the fold function *length*, and the resulting vector is a fused representation of the list and the classification proof. In the case of vectors, this classification is total and deterministic: every list is classified under one and only one index. But in general, classifications can be partial and nondeterministic. For example, promoting a list to an ordered list is classifying the list under an index that is a lower bound of the list. The classification process checks at each jump whether the list is still ordered; this check can fail, so an unordered list would "disappear" midway through the classification. Also there can be more than one lower bound for an ordered list, so the list can end up being classified under any one of them. Algebraic ornamentation in its original functional form can only capture part of this intuition about classification, namely those classifications that are total and deterministic. By generalising algebraic ornamentation to accept relational algebras, bringing in partiality and nondeterminacy, this idea about classification is captured in its entirety — a classification is just a relational fold computing the index that classifies an element. All ornaments specify classifications, and thus can be transformed into algebraic ornaments.

For more examples, let us first look at the classifying algebra for the ornament from natural numbers to lists. The base functor for natural numbers is

$$\mathbb{F}\ \mathit{NatD} \ :\ (\top \to \mathsf{Set}) \to (\top \to \mathsf{Set})$$
$$\mathbb{F}\ \mathit{NatD}\ X\ \_ \ =\ \Sigma\ \mathit{LTag}\ (\lambda\ \{\mathit{nil}-\mathit{tag} \to \top;\ \mathit{cons}-\mathit{tag} \to X\ \blacksquare\})$$

And the classifying algebra for the ornament *NatD-ListD A* is essentially

$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ :\ \mathbb{F}\ \mathit{NatD}\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\mathit{nil}-\mathit{tag}\quad ,\ \_\quad)\ (\mathsf{ok}\ \blacksquare)\ =\ \top$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\mathit{cons}-\mathit{tag}\ ,\ \mathsf{ok}\ t)\ (\mathsf{ok}\ \blacksquare)\ =\ A \times (t\ \equiv\ \blacksquare)$$

The result of folding a natural number n with this algebra is uninteresting, as it can only be $\mathsf{ok}\ \blacksquare$. The fold, however, requires an element of A for each successor node it encounters, so a proof that n goes through the fold consists of n elements of A. Another example is the ornament $OL\ =\ \lceil\ \mathit{OrdListOD}\ A\ \_\leqslant_A\_\ \rceil$ from lists to ordered lists, whose classifying algebra is essentially

$$\mathit{clsAlg}\ \mathit{OL}\ :\ \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ \mathit{OL}\ (\mathit{nil}-\mathit{tag}\quad ,\ \_\qquad)\ (\mathsf{ok}\ b)\ =\ \top$$
$$\mathit{clsAlg}\ \mathit{OL}\ (\mathit{cons}-\mathit{tag}\ ,\ a\ ,\ \mathsf{ok}\ b')\ (\mathsf{ok}\ b)\ =\ (b \leqslant_A a) \times (b'\ \equiv\ a)$$

In the nil case, the empty list can be mapped to any $\mathsf{ok}\ b$ because any $b\ :\ A$ is a lower bound of the empty list; in the cons case, where $a\ :\ A$ is the head and $\mathsf{ok}\ b'$ is a result of classifying the tail, i.e., $b'\ :\ A$ is a lower bound of the tail, the list can be mapped to $\mathsf{ok}\ b$ if $b\ :\ A$ is a lower bound of a and a is exactly b'.

Perhaps the most important consequence of the completeness theorem (in its present form) is that it provides a new perspective on the expressive power of ornaments and inductive families. We showed in a previous paper Ko and Gibbons [2013] that every ornament induces a promotion predicate and a corresponding family of isomorphisms (which is restated as (**??**) in **??**). But one question was untouched: can we determine (independently from ornaments) the range of predicates induced by ornaments? An answer to this question would tell us something about the expressive power of ornaments, and also about the expressive power of inductive families in general, since the inductive families we use are usually ornamentations of simpler algebraic datatypes from traditional functional programming. The completeness theorem offers such an answer: ornament-induced promotion predicates are exactly those expressible as relational folds (up to pointwise isomorphism). In other words, a predicate can be baked into a datatype by ornamentation if and only if it can be thought of as a nondeterministic classification of the elements of the datatype with a relational fold. This is more a guideline than a precise criterion, though, as the closest work about characterisation of the expressive power of folds discusses only functional folds Gibbons et al. [2001] (however, we believe that those results generalise to relations too). But this does encourage us to think about ornamentation computationally and to design new datatypes with relational algebraic methods. We illustrate this point with a solution to the **minimum coin change problem** in the next section.

6.1 Ornaments and horizontal transformations

6.2 Ornaments and relational algebras

6.3 Consequences

6.3.1 Parallel composition and the banana-split law

algebras corresponding to singleton ornaments and ornaments for optimised predicates

6.3.2 Ornamental algebraic ornaments

6.4 Discussion

bad computational behaviour; ornaments for optimised representation

Chapter 7

Conclusion

type computation — easy one like upgrades, swaps and more intricate one relying on universe construction; computational formalism — examples: ornaments, universal property of pullbacks; non-examples: relational calculus

7.1 Future work

fibred category theory for unifying similar categorical constructions; measure of representational efficiency; impact of homotopy type theory (e.g., functor equality); future of internalism (highly structural; scalability might lie in, e.g., hierarchical typing)

Bibliography

Thorsten ALTENKIRCH, James CHAPMAN, and Tarmo UUSTALU [2010]. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer-Verlag. doi: 10.1007/978-3-642-12032-9_21. ↰ page 111

Thorsten ALTENKIRCH and Conor MCBRIDE [2003]. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V. doi: 10.1007/978-0-387-35672-3_1. ↰ page 17

Thorsten ALTENKIRCH, Conor MCBRIDE, and James MCKINNA [2005]. Why dependent types matter. Available at http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf. ↰ page 2

Thorsten ALTENKIRCH, Conor MCBRIDE, and Wouter SWIERSTRA [2007]. Observational equality, now! In *Programming Languages meets Program Verification*, PLPV'07, pages 57–68. ACM. doi: 10.1145/1292597.1292608.

Thorsten ALTENKIRCH and Peter MORRIS [2009]. Indexed containers. In *Logic in Computer Science*, LICS'09, pages 277–285. IEEE. doi: 10.1109/LICS.2009.33. ↰ page 25

Robert ATKEY, Patricia JOHANN, and Neil GHANI [2012]. Refining inductive types. *Logical Methods in Computer Science*, 8(2:09). doi: 10.2168/LMCS-8(2:9)2012. ↰ pages 140 and 154

Gilles BARTHE, Venanzio CAPRETTA, and Olivier PONS [2003]. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293. doi: 10.1017/S0956796802004501. ↰ page 87

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 81, 83, and 153

Yves Bertot and Pierre Castéran [2004]. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag.

Richard Bird and Oege de Moor [1997]. *Algebra of Programming*. Prentice-Hall. ↰ pages 114, 118, and 130

Richard Bird and Jeremy Gibbons [2003]. Arithmetic coding with folds and unfolds. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag. doi: 10.1007/978-3-540-44833-4_1. ↰ pages 121 and 127

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag. ↰ page 2

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. doi: 10.1007/978-3-642-03153-3_2. ↰ page 2

Edwin Brady, Conor McBride, and James McKinna [2004]. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag. doi: 10.1007/978-3-540-24849-1_8. ↰ pages 19 and 28

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 17 and 52

Pierre-Évariste Dagand and Conor McBride [2012a]. Elaborating inductive definitions. arXiv:1210.6390. ↰ page 23

Pierre-Évariste Dagand and Conor McBride [2012b]. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM. doi: 10.1145/2364527.2364544. ↰ pages 17, 18, 36, 41, 51, 140, 153, and 154

Michael Dummett [2000]. *Elements of Intuitionism*. Oxford University Press, second edition.

Peter Dybjer [1998]. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549. doi: 10.2307/2586554. ↰ pages 17 and 152

Jeremy Gibbons [2007]. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139. doi: 10.1016/j.scico.2006.01.006. ↰ pages 121 and 125

Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch [2001]. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1):146–160. doi: 10.1016/S1571-0661(04)80906-X. ↰ page 144

Healfdene Goguen, Conor McBride, and James McKinna [2006]. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer-Verlag. doi: 10.1007/11780274_27.

Robert Harper and Robert Pollack [1991]. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136. doi: 10.1016/0304-3975(90)90108-T. ↰ page 87

Arend Heyting [1971]. *Intuitionism: An Introduction*. Amsterdam: North-Holland Publishing, third revised edition.

Hsiang-Shang Ko and Jeremy Gibbons [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi: 10.2201/NiiPi.2013.10.5. ↰ pages 18 and 143

Zhaohui Luo [1994]. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press.

Saunders Mac Lane [1998]. *Categories for the Working Mathematician*. Springer-Verlag, second edition. ↰ pages 85 and 99

Per Martin-Löf [1984a]. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London*, 312(1522):501–518. doi: 10.1098/rsta.1984.0073.

Per Martin-Löf [1984b]. *Intuitionistic Type Theory*. Bibliopolis, Napoli. ↰ pages 2, 3, and 17

Per Martin-Löf [1987]. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420.

Conor McBride [1999]. *Dependently Typed Functional Programs and their Proofs*. Ph.D. thesis, University of Edinburgh. ↰ page 90

Conor McBride [2003]. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075. doi: 10.1017/S0956796803004957.

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag. doi: 10.1007/11546382_3. ↰ pages 2 and 28

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ pages 24, 51, 118, 140, and 154

Conor McBride and James McKinna [2004]. The view from the left. *Journal of Functional Programming*, 14(1):69–111. doi: 10.1017/S0956796803004829. ↰ pages 8, 10, 11, and 135

Conor McBride and Ross Paterson [2008]. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13. doi: 10.1017/S0956796807006326. ↰ page 112

Erik MEIJER, Maarten FOKKINGA, and Ross PATERSON [1991]. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 124–144. Springer-Verlag. doi: 10.1007/3540543961_7. ↰ page 115

Eugenio MOGGI [1991]. Notions of computation and monads. *Information and Computation*, 93(1):55–92. doi: 10.1016/0890-5401(91)90052-4. ↰ page 111

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 50

Shin-Cheng MU, Hsiang-Shang KO, and Patrik JANSSON [2009]. Algebra of Programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579. doi: 10.1017/S0956796809007345. ↰ pages 89 and 120

Keisuke NAKANO [2013]. Metamorphism in jigsaw. *Journal of Functional Programming*, 23(2):161–173. doi: 10.1017/S0956796812000391. ↰ page 121

Bengt NORDSTRÖM [1988]. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619. doi: 10.1007/BF01941137. ↰ page 126

Bengt NORDSTRÖM, Kent PETERSON, and Jan M. SMITH [1990]. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Ulf NORELL [2007]. *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology. ↰ page 2

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. doi: 10.1007/978-3-642-04652-0_5. ↰ pages 2 and 7

Chris OKASAKI [1999]. *Purely functional data structures*. Cambridge University Press. ↰ pages 66, 67, 75, and 80

Christine PAULIN-MOHRING [1989]. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM. doi: 10.1145/75277.75285.

Tim SHEARD and Nathan LINGER [2007]. Programming in Ωmega. In *Central-European Functional Programming School*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer-Verlag. doi: 10.1007/978-3-540-88059-2_5.

Thomas STREICHER [1993]. Investigations into intentional type theory. Habilitation thesis, Ludwig Maximilian Universität.

Wouter SWIERSTRA [2008]. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. doi: 10.1017/S0956796808006758. ↰ page 52

THE UNIVALENT FOUNDATIONS PROGRAM [2013]. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton.

Philip WADLER [1987]. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313. ACM. doi: 10.1145/41625.41653.

Philip WADLER [1992]. The essence of functional programming. In *Principles of Programming Languages*, POPL'92, pages 1–14. ACM. doi: 10.1145/143165.143169. ↰ page 111

Todo list