Analysis and synthesis of inductive families

Hsiang-Shang Ko

28 February 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; a study not really emphasising practicality

Martin-Löf's intuitionistic type theory [Martin-Löf, 1975, 1984b; Nordström et al., 1990] 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.

Chapter 2

From intuitionistic type theory to dependently typed programming

This chapter serves three purposes.

- The general theme of the thesis is set up by describing the propositions-as-types-principle (Section 2.1) and its influence on program construction (Section 2.5).
- It is explained that, while we adopt AGDA as the expository language, we make sure that, at least in principle, every AGDA program in this thesis can be translated down to well-studied aspects of type theory — no mysterious AGDA-specific semantics is used. Specifically, we briefly discuss foundational aspects of pattern matching (Section 2.2) and equality (Section 2.3).
- Most of AGDA-specific syntax, notational conventions, and basic constructions used throughout the thesis are also introduced. In particular, a universe for "index-first" inductive families is constructed (Section 2.4), which is the basis of essentially all later constructions.

2.1 Propositions as types

Mathematics is all about mental constructions, that is, the intuitive grasp and manipulation of mental objects, the intuitionists say [Heyting, 1971; Dummett, 2000]. 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 (i.e., directly intuitively) 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 natural numbers in Martin-Löf Type Theory. In symbols, we are justified by our intuition to have the **formation rule**

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

saying that 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 inhabitants** of Nat, i.e., those inhabitants that are immediately recognisable as belonging to Nat, namely zero and suc n whenever n is an inhabitant of Nat. There are natural numbers which are not in canonical form (like 10^{10}) but instead encode an effective method for computing a canonical inhabitant. We accept them as **non-canonical inhabitants** 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 inhabitants, either directly or indirectly, as bearing a certain form and thus belonging to the set, so the inhabitants of the set are intuitively clear to us as a certain kind of mental constructions.

What is more characteristic of intuitionism is that the intuitionistic interpretation of propositions — in particular the logical constants/connectives — follows the same line of thought as the specification 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 exactly when 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 conjunctions

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

and the introduction rule

$$\frac{a : A \qquad b : B}{(a\, , b) : A \wedge B}$$

saying that an immediately acceptable proof (canonical inhabitant) of $A \wedge B$ is a pair of a proof (inhabitant) of A and a proof (inhabitant) of B. Any other (non-canonical) way of proving a conjunction must effectively yield a proof in the form of a pair. The relationship between a proposition and its proofs is thus exactly the same as the one between a set and its inhabitants — the proofs must be effectively recognisable as proving the proposition. Hence, in type theory, the notion of propositions and proofs is subsumed by the notion of sets and inhabitants. This is called the **propositions-as-types principle**, which reflects the observation that proofs are nothing but a certain kind of mental constructions.

Notice that the notion of "effective methods" — or computation — was presumed when the notion of sets 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 inhabitants, it is possible to express an effective method that mimics the construction of an inhabitant by saying that the computation has the same structure as how the inhabitant is constructed, and the computation is guaranteed to terminate since the structure of the inhabitant is finitary. For a typical example, let us look again at the natural numbers. Suppose that we have a **family of sets** $P : \mathsf{Nat} \rightarrow \mathsf{Set}$ indexed by inhabitants of Nat. (Since we only aim to present a casual sketch of type theory, we take the liberty of using AGDA functions (including Set-computing ones like P above) in places where terms under contexts should have been used.) If we have an inhabitant z of P zero and a method s that, for any $n : \mathsf{Nat}$, transforms an inhabitant of $P\ n$ to an inhabitant of $P\ (\mathsf{suc}\ n)$, then we can compute an inhabitant 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 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 structure of 2. This explanation justifies the following **elimination rule**

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

(The type of s illustrates AGDA's syntax for dependent function types — the value n of the first argument is referred to in the types of the second argument and the result.) The symbol $\mathsf{Nat\text{-}elim}$ symbolises the method described above, which, given P, z, and s, transforms every natural number n into an inhabitant of the set $P\ n$. The actual computation performed by $\mathsf{Nat\text{-}elim}$ is stated as two **computation rules** in the form of equality judgements (see Section 2.3):

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

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

From the logic perspective, predicates on Nat are a special case of Nat-indexed families of sets like P; $\mathsf{Nat\text{-}elim}$ then delivers the induction principle for natural numbers, as it produces a proof of $P\ n$ for every $n : \mathsf{Nat}$ if the base case z and the inductive case s can be proved. In general, the propositions-as-types principle treats logical entities as ordinary mathematical objects; the logic hence inherits the computational meaning of intuitionistic mathematics and becomes constructive.

By enabling the interplay of various sets governed by rules like the above ones, type theory is capable of formalising various mental constructions we manipulate in mathematics in a fully computational way, making it a powerful programming language. As Martin-Löf [1984a] 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". Indeed, sets are easily comparable with inductive datatypes in functional programming — a formation rule names a datatype, the associated introduction rules list the constructors of the datatype, and the associated elimination rule and computation rules define a precisely typed version of primitive recursion on the datatype. Consequently, we identify "sets" with "types" in this thesis, and regard them as interchangeable terms.

The uniform treatment of programs and proofs in type theory reveals new possibilities regarding proofs of program correctness. Traditional mathematical theories employ a standalone logic language for talking 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 (e.g., Hoare logic) then need to have a meta-level logic language to reason about properties of programs. In **dependently typed** languages based on type theory, however, 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 $R\ a\ b$, so the output of type B is guaranteed to be related to the input of type A by R. Checking of proof validity reduces to typechecking, and correctness proofs coexist with programs, as opposed to being separately presented at a meta-level.

The propositions-as-types principle, however, can lead to a more intimate form of program correctness by construction by blurring the distinction between programs and proofs even further; this form of program correctness — called **internalism** — is introduced in Section 2.5, which opens the central topic studied by this thesis. Before that, we make a transition from type theory to practical programming in AGDA, starting with its pattern matching notation.

2.2 Elimination and pattern matching

The formation rules and introduction rules for sets in type theory directly translate into inductive datatype declarations in functional programming. For example, the set of natural numbers is translated into AGDA as an inductive datatype with two constructors, with their full types displayed:

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

In type theory, computations on inductive datatypes are specified using eliminators like Nat-elim, whose style corresponds to **recursion schemes** [Meijer et al., 1991] in functional programming. One reason for making elimination as the only option is that programs in type theory are demanded to terminate — which is a consequence of the requirement that an inhabitant should be effectively recognisable as belonging to a set, and results in decidable typechecking — and using eliminators throughout is a straightforward way of enforcing termination. On the other hand, in functional programming, the **pattern matching** notation is widely used for defining programs on (inductive) datatypes (see, e.g., Hudak et al. [2007, Section 5]) in addition to recursion schemes. Pattern matching is vital to the clarity of functional programs because it not only allows a function to be intuitively defined by equations suggesting how the function computes, but also clearly conveys the programming strategy of splitting a problem into sub-problems by case analysis.

When it comes to dependently typed programming, the situation becomes more complicated due to the presence of **inductive families** [Dybjer, 1994], i.e., simultaneously inductively defined families of sets, like the following:

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

(An AGDA name with underscores (like $\_\leqslant_{\mathsf{N}}\_$) can be applied to arguments either normally (like "$\_\leqslant_{\mathsf{N}}\_ \ m\ n$") or by substituting the arguments for the underscores with proper spacing (like "$m \leqslant_{\mathsf{N}} n$").) Reading the declaration logically, the types of the two constructors refl and step give the two inference rules for establishing that one natural number is less than or equal to another. More generally, we read the declaration as a datatype parametrised by $m : \mathsf{Nat}$ (as signified by its appearance right next to **data** $\_\leqslant_{\mathsf{N}}\_$) and indexed by Nat. For any $m : \mathsf{Nat}$, the type family $\_\leqslant_{\mathsf{N}}\_ \ m : \mathsf{Nat} \to \mathsf{Set}$ as a whole is inductively populated: we have an inhabitant refl in the set $(\_\leqslant_{\mathsf{N}}\_ \ m)\ m$, and whenever we have an inhabitant $p : (\_\leqslant_{\mathsf{N}}\_ \ m)\ n$ for some $n : \mathsf{Nat}$, we can make a larger inhabitant $\mathsf{step}\ n\ p$ in another set $(\_\leqslant_{\mathsf{N}}\_ \ m)\ (\mathsf{suc}\ n)$ in the family. (From now on we usually refer to inductive families simply as datatypes, especially when emphasising their use in programming and de-emphasising the distinction between them and non-indexed datatypes like Nat.)

With inductive families, splitting a problem into sub-problems by case analysis in dependently typed programming often leads to nontrivial refinement of the goal type and the context, and such refinement can be tricky to handle with eliminators. Admittedly, in terms of expressive power, pattern matching and elimination are basically equivalent, as eliminators can be easily defined by dependent pattern matching, and conversely, it has been shown that dependent pattern matching can be reduced to elimination if **uniqueness of identity proofs** — or, equivalently, the **K axiom** [Streicher, 1993] — is assumed [McBride, 1999; Goguen et al., 2006]. (See Section 2.3 for more on uniqueness of identity proofs.) Nevertheless, there is a significant notational advantage of recovering pattern matching in dependently typed programming, especially with the support of an interactive development environment. Below we look at an example of interactively constructing a program with pattern matching in Agda, whose design was inspired by Epigram [McBride, 2004; McBride and McKinna, 2004].

2.2.1 Pattern matching and interactive development

Suppose that we are asked to prove that $\_\leqslant_{\mathsf{N}}\_$ is transitive, i.e., to construct the program

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

(The "telescopic" quantification "$(x\ y\ z : \mathsf{Nat}) \to$" is a shorthand for "$(x : \mathsf{Nat})\ (y : \mathsf{Nat})\ (z : \mathsf{Nat}) \to$", which, in turn, is a shorthand for "$(x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to (z : \mathsf{Nat}) \to$".) We define *trans* interactively by first putting pattern variables for the arguments on the left of the defining equation and then leaving an "interaction point" — also called a "goal" — on the right, which is numbered 0. Agda then tells us that a term of type $x \leqslant_{\mathsf{N}} z$ is expected (shown in the goal).

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

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

$$\begin{array}{llll}\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z & & & \\ \mathit{trans}\ x\ .z\ \ z & p^{\,x \leqslant_{\mathsf{N}} z}\ \mathsf{refl} & = & \{\, x \leqslant_{\mathsf{N}} z \,\}_1 \\ \mathit{trans}\ x\ \ y\ .(\mathsf{suc}\ w) & p^{\,x \leqslant_{\mathsf{N}} y}\ (\mathsf{step}\ w\ r^{\,y \leqslant_{\mathsf{N}} w}) & = & \{\, x \leqslant_{\mathsf{N}} \mathsf{suc}\ w \,\}_2\end{array}$$

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, upon enquiry, Agda tells us that the type of p in the context — which was originally $x \leqslant_{\mathsf{N}} y$ — is now $x \leqslant_{\mathsf{N}} z$ (shown in superscript next to p, which is not part of the Agda program but only appears when interacting with Agda). 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 sub-problems 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.

$$\begin{array}{lllll} \mathit{trans} : (x\,y\,z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\; x\; .z\;\; z & p & \mathsf{refl} & = & p \\ \mathit{trans}\; x\;\; y\; .(\mathsf{suc}\; w) & p^{\,x \leqslant_{\mathsf{N}} y} & (\mathsf{step}\; w\; r^{\,y \leqslant_{\mathsf{N}} w}) & = & \{\, x \leqslant_{\mathsf{N}} \mathsf{suc}\; w \,\}_2 \end{array}$$

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

$$\begin{array}{lllll} \mathit{trans} : (x\,y\,z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\; x\; .z\;\; z & p & \mathsf{refl} & = & p \\ \mathit{trans}\; x\;\; y\; .(\mathsf{suc}\; w) & p^{\,x \leqslant_{\mathsf{N}} y} & (\mathsf{step}\; w\; r^{\,y \leqslant_{\mathsf{N}} w}) & = & \mathsf{step}\; w\; \{\, x \leqslant_{\mathsf{N}} w \,\}_3 \end{array}$$

Now we see that the induction hypothesis term $\mathit{trans}\; x\; y\; w\; p\; r : x \leqslant_{\mathsf{N}} w$ has the right type. Filling the term into Goal 3 completes the program.

$$\begin{array}{llll} \mathit{trans} : (x\,y\,z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\; x\; .z\;\; z & p\; \mathsf{refl} & = & p \\ \mathit{trans}\; x\;\; y\; .(\mathsf{suc}\; w) & p\; (\mathsf{step}\; w\; r) & = & \mathsf{step}\; w\; (\mathit{trans}\; x\; y\; w\; p\; r) \end{array}$$

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

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

and write

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

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. Even with interactive support, the eliminator version 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.

2.2.2 Pattern matching on intermediate computation

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 convenient. McBride and McKinna [2004] thus proposed **with-matching**, which generalises pattern guards [Peyton Jones, 1997] and shifts pattern matching on intermediate computations from the right of an equation to the left, thereby granting them equal status with pattern matching on arguments, in particular the power to refine contexts and goal types.

Example (*insertion into a list*)**.** To demonstrate the syntax of **with**-matching, we give a simple example of writing the function inserting an element into a list as used in, e.g., insertion sort. (More precisely, the element is inserted at the rightmost position to the left of which all elements are strictly smaller.) First we define the usual list datatype:

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

and (throughout this thesis) let Val : Set be equipped with a decidable total ordering, i.e., there is a relation

$$\_{\leqslant}\_ : Val \to Val \to \mathsf{Set}$$

with the following operations:

$$\begin{array}{lll}
\leqslant\textit{-refl} & : & \{x : \mathit{Val}\} \to x \leqslant x \\
\leqslant\textit{-trans} & : & \{x\ y\ z : \mathit{Val}\} \to x \leqslant y \to y \leqslant z \to x \leqslant z \\
\leqslant?_ & : & (x\ y : \mathit{Val}) \to \mathsf{Dec}\ (x \leqslant y) \\
\not\leqslant\textit{-invert} & : & \{x\ y : \mathit{Val}\} \to \neg\ (x \leqslant y) \to y \leqslant x
\end{array}$$

where Dec is the following datatype witnessing whether a set is inhabited or not:

$$\begin{array}{l}
\mathbf{data}\ \mathsf{Dec}\ (A : \mathsf{Set}) : \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{yes} : \quad A \to \mathsf{Dec}\ A \\
\quad \mathsf{no}\ \ : \neg\ A \to \mathsf{Dec}\ A \quad \text{-- } \neg\ A = A \to \bot\text{, where } \bot \text{ is the empty set}
\end{array}$$

(Quantifications like $\{x : \mathit{Val}\}$ are implicit arguments to a function, which can be omitted when applying the function, and Agda would try to infer them.) The insertion function is then written as

$$\begin{array}{l}
\mathit{insert} : \mathit{Val} \to \mathsf{List}\ \mathit{Val} \to \mathsf{List}\ \mathit{Val} \\
\mathit{insert}\ y\ [\,] = y :: [\,] \\
\mathit{insert}\ y\ (x :: \mathit{xs})\ \mathbf{with}\ y \leqslant_? x \\
\mathit{insert}\ y\ (x :: \mathit{xs})\ \mid \mathsf{yes}\ _ = y :: x :: \mathit{xs} \\
\mathit{insert}\ y\ (x :: \mathit{xs})\ \mid \mathsf{no}\ \ _ = x :: \mathit{insert}\ y\ \mathit{xs}
\end{array}$$

The result of the intermediate computation $y \leqslant_? x : \mathsf{Dec}\ (y \leqslant x)$ is matched against yes and no on the left-hand side of the last two equations, just like we are performing pattern matching on a new argument. (In fact, Agda implements **with**-matching exactly by synthesising an auxiliary function with an additional argument [Norell, 2007, Section 2.3].) The witnesses carried by yes and no are ignored in this case (whose names are suppressed by underscores), but in general these can be proofs that are further matched and change the context and the goal type. (Admittedly, this is a trivial example with regard to the full power of **with**-matching, but *insert* will be used in later examples in this thesis.) □

An important application of **with**-matching is McBride and McKinna's adaptation [2004] of Wadler's **views** [1987] — or "customised pattern matching" — for dependently typed programming. Suppose that we wish to implement a snoc-list view for cons-lists, i.e., to say that a list is either empty or has the form $ys +\!\!+ (y :: [])$ (where $\_{+\!\!+}\_$ is list append, a definition of which is shown in Section 2.4.2). We would define the following view type

$$
\begin{array}{l}
\textbf{data}\ \mathsf{SnocView}\ \{A : \mathsf{Set}\} : \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{nil}\ \ \ : \mathsf{SnocView}\ [] \\
\quad \mathsf{snoc} : (ys : \mathsf{List}\ A)\ (y : A) \to \mathsf{SnocView}\ (ys +\!\!+ (y :: []))
\end{array}
$$

and write a **covering function** for the view:

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathit{snocView} : \{A : \mathsf{Set}\}\ (xs : \mathsf{List}\ A) \to \mathsf{SnocView}\ xs} \\
\multicolumn{5}{l}{\mathit{snocView}\ [] = \mathsf{nil}} \\
\mathit{snocView}\ (x :: xs) & \multicolumn{4}{l}{\textbf{with}\ \mathit{snocView}\ xs} \\
\mathit{snocView}\ (x :: .[]) & \mid & \mathsf{nil} & = & \mathsf{snoc}\ []\ x \\
\mathit{snocView}\ (x :: .(ys +\!\!+ (y :: []))) & \mid & \mathsf{snoc}\ ys\ y & = & \mathsf{snoc}\ (x :: ys)\ y
\end{array}
$$

Note that the type of *snocView* ensures that every list is covered under one constructor of SnocView. Also, this is a nontrivial example of **with**-matching, because performing pattern matching on the result of *snocView xs* refines *xs* in the context to either $[]$ or $ys +\!\!+ (y :: [])$, and the refinement is propogated to the goal type. Now, for example, the function *init* which removes the last element (if any) in a list can be implemented simply as

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathit{init} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A} \\
\mathit{init}\ xs & \multicolumn{4}{l}{\textbf{with}\ \mathit{snocView}\ xs} \\
\mathit{init}\ .[] & \mid & \mathsf{nil} & = & [] \\
\mathit{init}\ .(ys +\!\!+ (y :: [])) & \mid & \mathsf{snoc}\ ys\ y & = & ys
\end{array}
$$

Views are not enough for dependently typed programming, though, because views offer only customised case analyses but not terminating recursion. McBride and McKinna [2004] proposed a general mechanism for invoking any programmer-defined eliminator using the pattern matching syntax, so the programmer can choose whichever recursive problem-splitting strategy s/he needs and express it conveniently with pattern matching. This mechanism is implemented in EPIGRAM and greatly improves readability of dependently typed programs. Specifically, EPIGRAM syntax makes it clear which and in what order eliminators are invoked, so programs are easily guaranteed to be based on elimination — and thus are terminating — while remaining readable.

AGDA's design does not emphasise reducibility to elimination, but almost all the recursive programs in this thesis are written with this in mind, so termination is evident even without understanding AGDA's termination checker in detail. Also, although AGDA provides pattern matching only for datatype constructors, all but one of the recursive programs in this thesis use default structural induction (without need of programmer-defined elimination), so AGDA syntax suffices.

2.3 Equality

In logic, the **intension** of a concept is its defining description, while the **extension** of the concept is the range of objects it refers to. Two concepts can differ intensionally yet agree extensionally when they use different ways to describe the same range of objects. Classical mathematics cares about extensions only (e.g., the axiom of extensionality in set theory defines that two sets are equal exactly when they have the same inhabitants, regardless of how they are described), whereas in intuitionistic mathematics, objects are given to us as mental constructions, which are inherently intensional descriptions. As a consequence, the fundamental equality for intuitionistic mathematics is intensional [Dummett, 2000, Section 1.2], since we can only compare the intensional descriptions given to us, and furthermore it might be impossible to effectively recognise whether two intensionally different constructions are extensionally the same. For example, functions describing different computational procedures are distinguished even when they always map the same input to the same output, as it is well known that pointwise equality of functions is undecidable. We can, of course, still talk about extensional equalities in intuitionistic mathematics, but they are just treated as ordinary propositions.

The fundamental equality is formulated in type theory as **judgemental equality**, a meta-level notion for determining whether two types match in type-checking, which also involves determining whether two terms match because types can contain terms. (The computation rules for Nat-elim in Section 2.1 are examples of judgemental equalities between terms.) If we take the position of intuitionistic mathematics seriously, judgemental equality would be chosen to be the intensional, syntactic equality — also called **definitional equality** — which can be implemented by reducing two types/terms to normal forms and checking whether the normal forms match. The resulting type theory is called an **intensional type theory**, whose characteristic feature is decidable typechecking (enabling effective recognition of set membership) due to decidability of judgemental equality. AGDA, in particular, is intensional in this sense.

Judgemental equality — being a meta-level notion — is not an entity inside the theory. To state equality between two terms as a proposition and have proof for that proposition inside the theory, we need **propositional equality**, which can be defined in AGDA by the following inductive family:

$$\begin{array}{l} \textbf{data}\ \_\!\equiv\!\_\ \{A : \mathsf{Set}\}\ (x : A) : A \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{refl} : x \equiv x \end{array}$$

The canonical way to prove an equality proposition $x \equiv y$ is refl, which is permitted when x and y are judgementally equal. Under contexts, however, it is possible to prove that two judgementally different terms are propositionally equal. For example, the following "catamorphic" identity function on natural numbers

$$\begin{array}{lcl} id' : \mathsf{Nat} \to \mathsf{Nat} & & \\ id'\ \mathsf{zero} & = & \mathsf{zero} \\ id'\ (\mathsf{suc}\ n) & = & \mathsf{suc}\ (id'\ n) \end{array}$$

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

$$\begin{array}{l} id : \{A : \mathsf{Set}\} \to A \to A \\ id\ x\ =\ x \end{array}$$

That is, given $n : \mathsf{Nat}$ in the context, even though the two open terms $id\ n$ (which is definitionally just n) and $id'\ n$ are judgementally different, we can still prove $id\ n \equiv id'\ n$ by induction (elimination) on n, whose two cases instantiate n to a more specific form and make computation on $id'\ n$ happen. It might be said that propositional equality — in this most basic, inductive form — is "delayed" judgemental equality as a proposition: the judgementally different terms $id\ n$ and $id'\ n$ would compute to the same canonical term — and hence become judgementally equal — after substituting a canonical natural number for n, turning them into closed terms and allowing the computation to complete. Formally, this is stated as the "reflection principle": two propositionally equal closed terms are judgementally equal (see, e.g., Luo [1994, Section 5.1.3] and Streicher [1993, Section 1.1]).

A propositional equality satisfying the reflection principle — e.g., the AGDA one — can sometimes be too discriminating. For example, in category theory (which we use in Chapters 4 and 6), a universal function (i.e., a universal morphism in the category of sets and total functions) is unique up to extensional (i.e., pointwise) equality, but pointwise propositionally equal functions are usually not propositionally equal themselves. (If, under the empty context, two pointwise propositionally equal functions are propositionally equal themselves, then by the reflection principle they are also judgementally equal, but the two functions can well have different intensions.) Of course, we can explicitly work with pointwise equality on functions, i.e., establishing and using properties formulated in terms of the relation

$$\begin{aligned} &\_\doteq\_ : \{A\ B : \mathsf{Set}\} \to (A \to B) \to (A \to B) \to \mathsf{Set} \\ &\_\doteq\_\ \{A\}\ f\ g\ =\ (x : A) \to f\ x \equiv g\ x \end{aligned}$$

(The implicit argument A is explicitly displayed in curly braces on the left-hand side of the equation since we need to refer to I on the right-hand side.) Not being able to identify extensionally equal functions propositionally, however, means that we do not automatically get basic properties like

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

i.e., a function maps equal arguments to equal results, and

$$subst : \{A : \mathsf{Set}\}\ (P : A \to \mathsf{Set})\ \{x\ y : A\} \to x \equiv y \to P\ x \to P\ y$$

i.e., inhabitants in an indexed set can be transported to another set with an equal index, for extensionally equal functions. We would need to prove such properties for every entity like f and P on a case-by-case basis, which quickly becomes tedious.

Several foundational modifications to intensional type theory have been proposed to obtain a more liberal notion of equality. While this thesis sticks to AGDA's intensional approach and does not adopt any of the alternatives, it is interesting to reflect on the relationship between these alternatives and our development.

- A simple yet radical approach is to add the **equality reflection rule** to the theory, injecting propositional equality back into judgemental equality.

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

  Extensionally equal functions become judgementally equal (and thus propositionally equal) in such a theory: Suppose that 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, judgementally,

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

  The judgemental equality is thus able to identify pointwise equal functions and becomes extensional, and such a theory is called an **extensional type theory** (see, e.g., Nordström et al. [1990, Section 8.2]). In extensional type theory, combinators like *subst* become unnecessary, since having a proof of $x \equiv y$ means that x and y are identified judgementally and hence are regarded as the same during typechecking, so $P\ x$ and $P\ y$ are simply the same type — no explicit transportation is needed. Consequently, programs become very lightweight, with all the equality justifications moved to typing derivations at the meta-level. The downside is that typechecking in extensional type theory is undecidable, because whenever there is possibility that the equality reflection rule is needed, the typechecker would have to somehow determine whether there is a suitable equality proof, for which there is no effective procedure. This is not a big problem for proof assistants like NuPRL [Constable et al., 1985], in which the programmer instructs the proof assistant to construct typing derivations and can supply the right proof when using the equality reflection rule. (NuPRL, in fact, simply identifies judgemental equality and propositional equality and does not have the equality reflection rule explicitly.) But for programming languages like ΩMEGA [Sheard and Linger, 2007], equality reflection does present a problem, since the programmer constructs a term only, and the typing derivation has to be constructed by the typechecker, which then has to search for proofs. ΩMEGA can take hints from the programmer so the proof search is more likely to succeed, but the fundamental problem is that justification of program correctness now relies on the proof searching algorithm and is tied to the implementation detail of a specific programming system. Since the focus of this thesis is on dependently typed programming, extensional type theory is not a satisfactory foundation.

- Altenkirch et al. [2007] proposed a variant of intensional type theory called **observational type theory**, which defines propositional equality to be an extensional one — in particular, propositional equality on functions is pointwise equality — but retains computational behaviour (strong normalisation and canonicity) and decidable typechecking of intensional type theory. We step away from observational type theory merely for a practical reason: the theory is not implemented natively in any programming system yet. While it might be possible to model observational equality in AGDA to some extent and then construct the universes of descriptions (Section 2.4) and ornaments (Section 3.2) inside the observational model, developing and programming within the nested model would be too complex to be worthwhile.
- A new direction is being pursued by **homotopy type theory** [The Univalent Foundations Program, 2013], which gives propositional equality a higher-dimensional homotopic interpretation and broaden its scope with the univalence axiom and higher inductive types, but the computational meaning

of the theory remains an open problem. Its investigations into the higher dimensional structure of propositional equality might eventually lead to a systematic treatment of equality in dependently typed programming. For this thesis, however, we confine ourselves to a basic setting in which we freely invoke **uniqueness of identity proofs**, which is definable in AGDA by pattern matching:

$$\mathit{UIP} : \{A : \mathsf{Set}\}\ \{x\ y : A\}\ (p\ q : x \equiv y) \to p \equiv q$$
$$\mathit{UIP}\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}$$

Our identification of types and sets is thus consistent with the terminology of homotopy type theory, as types on which identity proofs are unique (and hence lack higher dimensional structure) are indeed termed "sets" by homotopy type theorists [The Univalent Foundations Program, 2013, Section 3.1].

With only AGDA's intensional equality, we have to explicitly work with equality-like propositions (like pointwise equality on functions) and manage them with the help of **setoids** [Barthe et al., 2003] in this thesis — Chapters 4 and 6, specifically. We will discuss to what extent the intensional approach works at the end of these chapters.

2.4 Universes and datatype-generic programming

Martin-Löf [1984b] introduced the notion of **universes** to support "large elimination", i.e., arbitrary computation of sets, which is necessary for proving the fourth Peano axiom that zero is not the successor of any natural number [Smith, 1988]. A universe (à la Tarski) is a set of codes for sets, which is equipped with a decoding function translating codes to sets. Large elimination is then computing an inhabitant in the universe (via ordinary elimination like Nat-elim) and decoding the result to a set. Another important purpose of universes is to support quantification over sets while precluding paradoxical formation of self-referential sets — AGDA, for example, has a universe hierarchy (Set, Set_1, Set_2, $\ldots$) for this purpose. From the programming perspective, however, the most interesting case is when the universe is supplied with an elimination rule [Nordström et al., 1990, Section 14.2]: allowing computation on universes turns out to correspond to **datatype-generic programming** [Gibbons, 2007a], whose idea is that the "shapes" of datatypes can be encoded so programs can be defined in terms of these shapes. Universes can encode such shapes, and since universes are just ordinary sets, datatype-generic programming becomes just ordinary programming in dependently typed languages [Altenkirch and McBride, 2003].

In this thesis we not only use but also need to compute a class of inductive families which we call **index-first datatypes** [Chapman et al., 2010; Dagand and McBride, 2012b], and hence need to construct a universe for them. Before that, we give a high-level introduction to these datatypes first.

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. For example, consider the following datatype of **vectors**, i.e., length-indexed lists:

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

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, distinguishing it from an Agda datatype declaration. 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 $A : \mathsf{Set}$:

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

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

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

$\mathsf{Vec}\ A$ 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:

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

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. □

2.4.2 Universe construction

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

$$
\begin{array}{l}
VecF\ A : (\mathsf{Nat} \to \mathsf{Set}) \to (\mathsf{Nat} \to \mathsf{Set}) \\
VecF\ A\ X\ \mathsf{zero} \qquad = \top \\
VecF\ A\ X\ (\mathsf{suc}\ n) = A \times X\ n \quad \text{-- \_×\_ is cartesian product}
\end{array}
$$

and take its least fixed point. ($\top$ is a singleton set whose only inhabitant is "$\blacksquare$".) If we flip the order of the arguments of $VecF\ A$:

$$
\begin{array}{l}
VecF'\ A : \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Set} \\
VecF'\ A\ \mathsf{zero} \qquad = \lambda\ X \to \top \\
VecF'\ A\ (\mathsf{suc}\ n) = \lambda\ X \to A \times X\ n
\end{array}
$$

we see that $\mathit{VecF}'\ A$ consists of two different "responses" to the index request, each of type $(\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Set}$. 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:

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

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

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

with decoding function

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

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:

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

where $\_\rightrightarrows\_$ is defined by

$$\begin{aligned}&\_\rightrightarrows\_ : \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ &\_\rightrightarrows\_\ \{I\}\ X\ Y = \{i : I\} \to X\ i \to Y\ i\end{aligned}$$

Remark (*presentation of universes and their decoding*). We always present universes (e.g., RDesc) along with their decoding (e.g., $[\![\_]\!]$ for RDesc) to emphasise the meaning of the codes, even when the decoding is not logically tied to the codes (cf. Martin-Löf's universe [1984b], which is inductive-recursive [Dybjer, 1998] and must present the universe and its decoding simultaneously). □

Notation (*dependent pairs and* Agda *records*). Cartesian product is a special case of **Σ-types**, also known as **dependent pairs**, which are defined in AGDA as a record:

$$
\begin{array}{l}
\textbf{record}\ \Sigma\ (A\ :\ \mathsf{Set})\ (X\ :\ A \to \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ \_,\_ \\
\quad \textbf{field} \\
\quad\quad \mathit{outl}\ :\ A \\
\quad\quad \mathit{outr}\ :\ X\ \mathit{outl} \\
\textbf{infixr}\ 4\ \_,\_ \\
\textbf{open}\ \Sigma \\
\textbf{syntax}\ \Sigma\ A\ (\lambda\ a \to T)\ =\ \Sigma[\,a : A\,]\ T
\end{array}
$$

An inhabitant of $\Sigma\ A\ X$ is a pair where the type of the second component depends on the first component, and is written by listing values for the fields like

$$
\begin{array}{l}
\textbf{record}\ \{\ \mathit{outl}\ =\ a \\
\qquad\quad ;\ \mathit{outr}\ =\ x\ \}
\end{array}
$$

(where $a\ :\ A$ and $x\ :\ X\ a$) — a cartesian product $A \times B$ is thus a special case, which is defined as $\Sigma\ A\ (\lambda\ \_\ \mapsto\ B)$. The **constructor** declaration gives rise to a constructor function

$$\_,\_\ :\ \{A\ :\ \mathsf{Set}\}\ \{X\ :\ A \to \mathsf{Set}\} \to (a\ :\ A) \to X\ a \to \Sigma\ A\ X$$

which associates to the right when used as an infix operator because of the **infixr** statement below, and can be used in pattern matching. The two field declarations give rise to two projection functions, qualified by "Σ.":

$$
\begin{array}{l}
\Sigma.\mathit{outl}\ :\ \{A\ :\ \mathsf{Set}\}\ \{X\ :\ A \to \mathsf{Set}\} \to \Sigma\ A\ X \to A \\
\Sigma.\mathit{outr}\ :\ \{A\ :\ \mathsf{Set}\}\ \{X\ :\ A \to \mathsf{Set}\} \to (p\ :\ \Sigma\ A\ X) \to X\ (\Sigma.\mathit{outl}\ p)
\end{array}
$$

We can drop the qualifications and refer to them simply as *outl* and *outr* due to the **open** statement. Finally, we can treat Σ as a binder and write, e.g., $\Sigma\ A\ X$ as $\Sigma[\,a : A\,]\ X\ a$, due to the **syntax** statement. □

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

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

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

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

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

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 informally refer to the index part of a σ as a **field** of the datatype. Like Σ, we sometimes 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 $\mathsf{Desc}\ \top$.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{ListTag}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \text{'}\mathsf{nil}\ \ :\ \mathsf{ListTag} \\
\quad \text{'}\mathsf{cons}\ :\ \mathsf{ListTag}
\end{array}
$$

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

The index request is necessarily $\blacksquare$, and we respond with a field of type $\mathsf{ListTag}$ representing the constructor choices. A pattern-matching lambda function (which is syntactically distinguished by enclosing its body in curly braces) follows, which computes the trailing responses to the two possible values 'nil and 'cons for the field: if the field receives ‘nil, then we are constructing zero, which takes no recursive values, so we write v [] to end this branch; if the ListTag field receives ‘cons, then we are constructing a successor, which takes a recursive value at index ▪, so we write v (▪ :: []). □

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.

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

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. □

Example (*vectors*). The datatype of vectors is parametrised by the element type and (nontrivially) 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 this section. □

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:

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

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

List append can then be rewritten in the usual form (assuming that the terms $[\,]$ and $\_{::}\_$ can be used in pattern matching, which is not actually allowed in Agda, though):

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

Later on, encoded datatypes are almost always accompanied by corresponding index-first datatype declarations, which are 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:

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

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

mutual

$$\begin{array}{l}
\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X) \\
\mathit{fold}\ \{I\}\ \{D\}\ f\ \{i\}\ (\mathsf{con}\ ds) = f\ (\mathit{mapFold}\ D\ (D\ i)\ f\ ds)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mapFold} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ (D' : \mathsf{RDesc}\ I) \to} \\
\multicolumn{3}{l}{\qquad\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to [\![\ D'\ ]\!]\ (\mu\ D) \to [\![\ D'\ ]\!]\ X} \\
\mathit{mapFold}\ D\ (\mathsf{v}\ [])\quad f\ \blacksquare & = & \blacksquare \\
\mathit{mapFold}\ D\ (\mathsf{v}\ (i :: is))\ f\ (d\ ,\ ds) & = & \mathit{fold}\ f\ d\ ,\ \mathit{mapFold}\ D\ (\mathsf{v}\ is)\ f\ ds \\
\mathit{mapFold}\ D\ (\sigma\ S\ D')\quad f\ (s\ ,\ ds) & = & s\ ,\ \mathit{mapFold}\ D\ (D'\ s)\ f\ ds
\end{array}$$

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

to define *fold* by

$$\begin{array}{l}
\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X) \\
\mathit{fold}\ \{I\}\ \{D\}\ f\ \{i\}\ (\mathsf{con}\ ds) = f\ (\mathit{mapRD}\ (D\ i)\ (\mathit{fold}\ f)\ ds)
\end{array}$$

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

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mapRD} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I) \to} \\
\multicolumn{3}{l}{\qquad\qquad \{X\ Y : I \to \mathsf{Set}\}\ (g : X \rightrightarrows Y) \to [\![\ D\ ]\!]\ X \to [\![\ D\ ]\!]\ Y} \\
\mathit{mapRD}\ (\mathsf{v}\ [])\quad g\ \blacksquare & = & \blacksquare \\
\mathit{mapRD}\ (\mathsf{v}\ (i :: is))\ g\ (x\ ,\ xs) & = & g\ x\ ,\ \mathit{mapRD}\ (\mathsf{v}\ is)\ g\ xs \\
\mathit{mapRD}\ (\sigma\ S\ D)\quad g\ (s\ ,\ xs) & = & s\ ,\ \mathit{mapRD}\ (D\ s)\ g\ xs
\end{array}$$

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 to AGDA, 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.

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 in Chapter 3). 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 Externalism and internalism

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 Σ-types of dependent pairs. In a type $\Sigma\ A\ P$, when A is interpreted as a ground set and P as a predicate on A, an inhabitant of $\Sigma\ A\ P$ is an inhabitant a of A paired with a proof that $P\ a$ holds. When programs are the objects we reason about, this style naturally suggests a 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. 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 type theory of subsets was given 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] uses a type-theoretic foundation [Coquand and Huet, 1988; Coquand and Paulin-Mohring, 1990] which distinguishes programs and proofs, and proofs are written in a tactic-based language, differently from programs; it is also famous for the ability to extract executable portions of proof scripts into programs [Paulin-Mohring, 1989; Letouzey, 2003], as used in the CompCert project developing a verified C compiler [Leroy, 2009], for example.

On the other hand, having unified programs and proofs in type theory (Section 2.1), it seems a pity if the unification is not exploited to a deeper level. In Dijkstra's proposal for program correctness by construction, proofs and programs should be conceived "hand in hand", and the unification of programs and proofs brings us unprecedentedly closer to this ideal, since we can start thinking about programs that also serve as correctness proofs themselves. The dependently typed programming community has been exploring the use of inductive families not only for defining predicates on data (like $\_\leqslant_{\mathsf{N}}\_$) but also for representing data with embedded constraints (like Vec). Programs manipulating such datatypes would also deal with the embedded constraints and are thus correct by construction. Vector append is a classic (albeit somewhat trivial) example: Defining addition on natural numbers as

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

vector append is then defined by

$$
\begin{aligned}
&\_+\!\!+\_ \;:\; \{A \;:\; \mathsf{Set}\}\; \{m\; n \;:\; \mathsf{Nat}\} \to \mathsf{Vec}\; A\; m \to \mathsf{Vec}\; A\; n \to \mathsf{Vec}\; A\; (m + n)\\
&[\,] \qquad\quad +\!\!+\; ys \;=\; ys\\
&(x :: xs) +\!\!+\; ys \;=\; x :: (xs +\!\!+\; ys)
\end{aligned}
$$

The program for vector append looks exactly like the one for list append except for the more informative type, which makes it possible for the program to meet its specification — the length of the result of append should be the sum of the lengths of the two input lists — by construction. For list append, whose type uses the plain list datatype, we need to separately produce the following proof:

$$
\begin{aligned}
&\mathit{append\text{-}length} \;:\\
&\quad \{A \;:\; \mathsf{Set}\}\; (xs\; ys \;:\; \mathsf{List}\; A) \to \mathit{length}\; (xs +\!\!+\; ys) \;\equiv\; \mathit{length}\; xs + \mathit{length}\; ys\\
&\mathit{append\text{-}length}\; [\,] \qquad\quad ys \;=\; \mathsf{refl}\\
&\mathit{append\text{-}length}\; (x :: xs)\; ys \;=\; \mathit{cong}\; \mathsf{suc}\; (\mathit{append\text{-}length}\; xs\; ys)
\end{aligned}
$$

But by switching to the vector datatype, the proof dissolves into the typing of the program and needs no separate handling. This is possible because list append and the proof *append-length* share the same structure; consequently, by careful type design, the vector append program alone is able to carry both of them simultaneously. Ko and Gibbons [2011] proposed to call this programming style **internalism**, suggesting that proofs are internalised in programs, while the traditional proving-after-programming style is called **externalism**. Externalism is necessary when we wish to show that an existing program satisfies additional properties, especially when the proofs are complicated and do not follow the structure of the program. On the other hand, writing a (simply typed) program and an externalist proof following the structure of the program is like stating the same thing twice: even though the programmer knows the meaning of the program, s/he has to first state the meaningless symbol manipulation aspect and then explain its meaning via a separate proof, doubling the effort. In contrast, internalism is programming with informative types so as to give precise description of meaningful computations, so explanations via separate proofs become unnecessary. As McBride [2004] aptly put it, internalism makes programs and their explanations via proofs "not merely coexist but coincide". In addition, by encoding meanings in types, semantic considerations are (at least partially) reduced to syntax and can be aided mechanically — Agda's interactive development environment (Section 2.2.1) is one form of such aid. With interactive development, internalist types not only passively rule out nonsensical programs, but can actively provide helpful information to guide program construction. We will see several examples of such "type-directed programming" in this thesis (notably in Section 5.3).

Historical remark. The distinction between internalism and externalism at least goes back to simply typed λ-calculus á la Church and Curry — a Church-style λ-term always appears with its type and they together denote a value of the type, whereas Curry-style λ-terms are untyped and all reside in a common domain, and a term is later stated to have additional properties via typing. Reynolds [2000] called the two styles as giving "intrinsic" and "extrinsic" semantics to a language, and established relationship between the two kinds of semantics via a logical relations theorem. Our emphasis is on a more practical aspect about how intrinsic and extrinsic semantics can affect the attitude and approach to program construction. □

Internalism comes with its own problems, however. For one: internalist type design is difficult, yet there is almost no effective guideline or discipline for such design. Carelessly designed internalist types can lead to less natural programs. A simple example is when we switch the order of the arguments of the addition in the type of vector append,

$$\_\!+\!\!+\_ : \{A : \mathsf{Set}\}\ \{m\ n : \mathsf{Nat}\} \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (n + m)$$

$$\begin{array}{lcl} [\,] \quad\quad\; +\!\!+ \; ys & = & subst \; (\mathsf{Vec}\; A) \; \{\, n \;\equiv\; n + \mathsf{zero} \,\}_0 \; ys \\ (x :: xs) +\!\!+ \; ys & = & subst \; (\mathsf{Vec}\; A) \; \{\, \mathsf{suc}\; (n + m) \;\equiv\; n + \mathsf{suc}\; m \,\}_1 \; (x :: (xs +\!\!+ \; ys)) \end{array}$$

we would be forced to perform two type-casts that could have been avoided. Not only does the program look ugly, but this can also cause a cascade effect when we need to write other programs whose types depend on this program (e.g., externalist proofs about it), which would have to deal with the type-casts. McBride [2012] gave a more interesting example: to prove the polynomial testing principle (two polynomial functions of degree n are pointwise equal if they agree at $n + 1$ different points), McBride started with a datatype of encodings of polynomial functions indexed by degree but, after trying to program with the datatype, quickly found out that the datatype should instead be indexed by an arbitrary upper bound of the degree so a relaxed form of the polynomial testing principle can be naturally programmed (two polynomial functions of degree <u>at most</u> n are pointwise equal if they agree at $n + 1$ different points). Scalability is another issue, especially when the only tool we have is the primitive language of datatype declarations: writing a datatype declaration with a sophisticated property internalised is comparable to programming a sophisticated algorithm in assembly language, and understanding the meaning of a complicated datatype declaration takes thorough reading and some inductive guessing and reasoning. In short, the complexity of internalist types has made type design a nontrivial programming problem, and we are in serious lack of type-level programming support.

Internalist library design also poses a problem: Since internalism requires differently indexed versions of the same data structure, an internalist library should provide more or less the same set of operations for all possible variants of the data structure. Without a way to manage these formally unrelated datatypes and operations modularly, an ad hoc library would need to duplicate the same structure and logic for all the variants and becomes hard to expand. For example, suppose that we have constructed a library for lists that include vectors, ordered lists, and ordered vectors, and now wish to add a new flavour of lists, say, association lists indexed with the list of keys. Operations need to be reimplemented not only for such key-indexed lists, but also for key-indexed vectors, ordered key-indexed lists, and ordered key-indexed vectors, even though key-indexing is the only new feature. An ideal structure for such a library would be having a separate module for each of the properties about length, ordering, and key-indexing. These modules can be developed independently, and there would be a way to assemble components in these modules at will — for example, ordered vectors and related operations would be synthesised from the components in the modules about length and ordering. This ideal library structure calls for some form of composability of internalist datatypes and operations.

Composability has never been a problem for externalism, however. In an externalist list library, we would have only one basic list datatype and several predicates on lists about length, ordering, key-indexing, etc. Lists are "promoted" to vectors, ordered lists, or ordered vectors by simply pairing the list datatype with the length predicate, the ordering predicate, or the pointwise conjunction of the two predicates, respectively. Common operations are implemented for basic lists only, and their properties regarding length or ordering are proved independently and invoked when needed. Can we somehow introduce this beneficial composability to internalism as well? The answer is yes, because there are isomorphisms between externalist and internalist datatypes to be exploited.

To illustrate, let us go through a small case study about upgrading the *insert* function on lists (Section 2.2.2) for vectors, ordered lists, and ordered vectors. The externalist would define vectors as a Σ-type,

$$
\begin{aligned}
&\mathit{ExtVec} : \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathit{ExtVec}\ A\ n \ = \ \Sigma[\,\mathit{xs} : \mathsf{List}\ A\,]\ \mathit{length}\ \mathit{xs} \equiv n
\end{aligned}
$$

prove that *insert* increases the length of a list by one,

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

and define insertion on vectors as

$$
\begin{aligned}
&\mathit{insert}_{EV} : \mathit{Val} \to \{n : \mathsf{Nat}\} \to \mathit{ExtVec}\ \mathit{Val}\ n \to \mathit{ExtVec}\ \mathit{Val}\ (\mathsf{suc}\ n) \\
&\mathit{insert}_{EV}\ y\ (\mathit{xs}\ ,\ \mathit{len}) \ = \ \mathit{insert}\ y\ \mathit{xs}\ ,\ \mathit{insert\text{-}length}\ y\ \mathit{xs}\ \mathit{len}
\end{aligned}
$$

which processes the list and the length proof by *insert* and *insert-length* respectively. Similarly for ordered lists (indexed with a lower bound), the externalist would use the Σ-type

$$\begin{aligned}
&\mathit{ExtOrdList} \;:\; \mathit{Val} \to \mathsf{Set} \\
&\mathit{ExtOrdList}\ b \;=\; \Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,]\ \mathsf{Ordered}\ b\ \mathit{xs}
\end{aligned}$$

where the Ordered predicate is defined by

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

Insertion on ordered lists is then

$$\begin{aligned}
&\mathit{insert}_{EO} \;:\; (y \;:\; \mathit{Val})\ \{b \;:\; \mathit{Val}\} \to \mathit{ExtOrdList}\ b \to \\
&\qquad\qquad\ \{b' \;:\; \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \mathit{ExtOrdList}\ b' \\
&\mathit{insert}_{EO}\ y\ (\mathit{xs}\ ,\ \mathit{ord})\ b'{\leqslant}y\ b'{\leqslant}b \;=\; \mathit{insert}\ y\ \mathit{xs}\ ,\ \mathit{insert\text{-}ordered}\ y\ \mathit{xs}\ \mathit{ord}\ b'{\leqslant}y\ b'{\leqslant}b
\end{aligned}$$

where *insert-ordered* proves that *insert* preserves ordering:

$$\begin{aligned}
&\mathit{insert\text{-}ordered} \;:\; (y \;:\; \mathit{Val})\ \{b \;:\; \mathit{Val}\}\ (\mathit{xs} \;:\; \mathsf{List}\ \mathit{Val}) \to \mathsf{Ordered}\ b\ \mathit{xs} \to \\
&\qquad\qquad\qquad\quad \{b' \;:\; \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \mathsf{Ordered}\ b'\ (\mathit{insert}\ y\ \mathit{xs})
\end{aligned}$$

Now the externalist has arrived at a modular list library (albeit a tiny one), which contains

- a basic module consisting of the basic list datatype and insertion on basic lists, and
- two independent upgrading modules about length and ordering, each consisting of a predicate on lists and a related proof about insertion.

It is easy to mix all three modules and get ordered vectors and insertion on them. The Σ-type uses the pointwise conjunction of the two predicates,

$$\begin{aligned}
&\mathit{ExtOrdVec} \;:\; \mathit{Val} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathit{ExtOrdVec}\ b\ n \;=\; \Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,]\ \mathsf{Ordered}\ b\ \mathit{xs} \times \mathit{length}\ \mathit{xs} \;\equiv\; n
\end{aligned}$$

and insertion simply uses *insert-ordered* and *insert-length* to process the two proofs bundled with a list:

$$\begin{aligned}
&\mathit{insert}_{EOV} : (y : \mathit{Val})\ \{b : \mathit{Val}\}\ \{n : \mathsf{Nat}\} \to \mathit{ExtOrdVec}\ b\ n \to \\
&\qquad\qquad \{b' : \mathit{Val}\} \to b' \leqslant y \to b' \leqslant b \to \mathit{ExtOrdVec}\ b'\ (\mathsf{suc}\ n) \\
&\mathit{insert}_{EOV}\ y\ (\mathit{xs}\ ,\ \mathit{ord}\ ,\ \mathit{len})\ b'{\leqslant}y\ b'{\leqslant}b\ =\ \mathit{insert}\ \ y\ \mathit{xs}\ , \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{insert\text{-}ordered}\ y\ \mathit{xs}\ \mathit{ord}\ b'{\leqslant}y\ b'{\leqslant}b\ , \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{insert\text{-}length}\ \ y\ \mathit{xs}\ \mathit{len}
\end{aligned}$$

This is the kind of library we are looking for, except that the types are all externalist. The externalist and internalist types are not unrelated, however. For example, internalist and externalist vectors are related by the indexed family of **conversion isomorphisms**:

$$\mathit{Vec\text{-}iso}\ A : (n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \cong \mathit{ExtVec}\ A\ n$$

Fixing $n : \mathsf{Nat}$, the left-to-right direction of the isomorphism

$$\mathsf{Iso}.\mathit{to}\ (\mathit{Vec\text{-}iso}\ A\ n) : \mathsf{Vec}\ A\ n \to \Sigma[\,\mathit{xs} : \mathsf{List}\ A\,]\ \mathit{length}\ \mathit{xs} \equiv n$$

computes the underlying list of a vector and a proof that the list has length n, and the right-to-left direction

$$\mathsf{Iso}.\mathit{from}\ (\mathit{Vec\text{-}iso}\ A\ n) : (\Sigma[\,\mathit{xs} : \mathsf{List}\ A\,]\ \mathit{length}\ \mathit{xs} \equiv n) \to \mathsf{Vec}\ A\ n$$

promotes a list to a vector when there is a proof that the list has length n. (The definition of $\_{\cong}\_$, which is actually an instance of a record datatype Iso, appears in Section 4.1.) To get insertion on internalist vectors, we convert the input vector to its externalist representation, make insert_{EV} do the work, and convert the result back to the internalist representation; more formally, the operation

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

is defined by the commutative diagram:

$$\begin{tikzcd}
\mathsf{Vec}\ \mathit{Val}\ n \arrow[rr, "\mathit{insert}_V\ y"] \arrow[d, "\mathsf{Iso}.\mathit{to}\ (\mathit{Vec\text{-}iso}\ \mathit{Val}\ n)"'] & & \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n) \\
\begin{array}{c}\Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,] \\ \mathit{length}\ \mathit{xs} \equiv n\end{array} \arrow[rr, "\mathit{insert}\ y\ *", "\mathit{insert\text{-}length}\ y\ _"'] & & \begin{array}{c}\Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,] \\ \mathit{length}\ \mathit{xs} \equiv \mathsf{suc}\ n\end{array} \arrow[u, "\mathsf{Iso}.\mathit{from}\ (\mathit{Vec\text{-}iso}\ \mathit{Val}\ (\mathsf{suc}\ n))"']
\end{tikzcd}$$

(The $\_{*}\_$ operator is defined by $(f * g)\ (x\ , y) \ = \ (f\ x\ , g\ y)$, and the underscore leaves out the term for AGDA to infer.) Similarly, we can get insertion for internalist ordered lists and ordered vectors (definitions to appear in Chapter 3) from the externalist library by suitable conversion isomorphisms of the same form as *Vec-iso*. It is due to these conversion isomorphisms between internalist and externalist representations that we can **analyse** internalist datatypes into externalist components, which can then be modularly processed. This analysis of internalist datatypes and its application to modular library structuring is explored in Chapter 3 (in particular, the insertion example is resolved in Section 3.4.1).

The interconnection between internalism and externalism (in the form of conversion isomorphisms) also shed some light on supporting internalist type design. The **synthetic** direction of the interconnection goes from basic types and predicates to internalist types. It is conceivable that, for externalist predicates of some particular form, we can manufacture corresponding internalist types on the other side of the interconnection. The externalist side of the interconnection is usually kept non-dependently typed, so it is possible to use existing non-dependently typed calculi to derive suitable externalist predicates from specifications, which are then used to manufacture datatypes on the internalist side for type-directed programming. Chapter 5 presents one such approach, using relational calculus [Bird and de Moor, 1997] as a design language for internalist datatypes. Rather than improvising internalist types and hoping that they will work, we write specifications in the form of relational programs, which are amenable to algebraic transformation and can be much more concise and readable than the language of datatype declarations, making it easier to arrive at helpful and comprehensible internalist types.

To sum up: While internalism offers type-directed program construction and reduces the burden of producing correctness proofs, additional or more complicated properties of existing programs can only be established by externalism, which naturally gives rise to a modular organisation. Both internalism and externalism are here to stay, since the two styles serve different purposes and have their own pros and cons. Exploiting their interconnection can lead to interesting programming patterns, which the rest of this thesis explores.

Chapter 3

Refinements and ornaments

This chapter begins our exploration of the interconnection between internalism and externalism by looking at **the analytic direction**, i.e., the decomposition of a sophisticated datatype into a basic datatype and a predicate on the basic datatype. More specifically, we assume that the sophisticated datatype and the basic datatype are known and their descriptions (Section 2.4.2) are straightforwardly related by an **ornament** (Section 3.2), and derive from the ornament an externalist predicate and an indexed family of conversion isomorphisms. As discussed in Section 2.5, one purpose of such decomposition is for internalist datatypes and operations to take a round trip to the externalist world so as to harvest composability there. The task can be broken into two parts:

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

Refinements (Section 3.1), which axiomatise conversion isomorphisms between internalist and externalist datatypes, are the abstraction we introduce for bridging the two parts. The first part is then formalised with **upgrades** (Section 3.1.2) which use refinements as their components, and the second part is done by translating ornaments to refinements (Section 3.3). To actually harvest externalist composability, we need conversion isomorphisms in which the externalist predicates involved are pointwise conjunctions (as in the case of externalist ordered vectors in Section 2.5). Such conversion isomorphisms come from **parallel composition** of ornaments (Section 3.2.3), which not only gives rise to pointwise conjunctive predicates on the externalist side (Section 3.3.2) but also produces composite datatypes on the internalist side (e.g., the internalist datatype of ordered vectors incorporating both ordering and length information). The above framework of ornaments, refinements, and upgrades are illustrated with several examples in Section 3.4, followed by some discussion (including related work) in Section 3.5.

3.1 Refinements

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

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

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

3.1.1 Refinements between individual types

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

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

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

Example (*refinement from lists to ordered lists*). Consider the internalist datatype of ordered lists (indexed by a lower bound; the type *Val* and associated operations are postulated in Section 2.2.2):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Coherence-based definition of refinements

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

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

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

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

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, can be more easily generalised for function types, as we will see below.

3.1.2 Upgrades

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

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

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

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

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

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

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

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

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

That is, we decompose the input list into the underlying natural number (i.e., its length) and a vector of elements, process the two parts separately with double and $\mathit{duplicate}'$, and finally combine the results back to a list. (This is what we did for insert_V in Section 2.5.) The relationship between the promoted function *duplicate* and the underlying function *double* is characterised by the **coherence property**

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

or as a commutative diagram:

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

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

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

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

As an Agda record datatype:

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

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

Upgrades from refinements

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

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

and consequently any refinement gives rise to an upgrade:

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

Composition of upgrades

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

$$
\begin{array}{l}
\_\rightharpoonup\_ \;:\; \{A\;A'\;B\;B' \;:\; \mathsf{Set}\} \to \\
\qquad\qquad \mathsf{Refinement}\;A\;A' \to \mathsf{Upgrade}\;B\;B' \to \mathsf{Upgrade}\;(A \to B)\;(A' \to B')
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

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

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

and the upgrade derived from r by *toUpgrade*, we get an upgrade

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

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

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

3.1.3 Refinement families

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

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

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

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

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

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

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

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

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

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

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

This further classification is captured as a refinement family of type

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

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

Refinement families are the vehicle we use to express conversion relationship between inductive families. For now, however, they have to be prepared manually, which requires considerable effort. Also, when it comes to acquiring externalist composability for internalist datatypes, we need to be able to compose refinements such that the promotion predicate of the resulting refinement is the pointwise conjunction of existing promotion predicates, so we get conversion isomorphisms of the right form. For example, we should be able to compose the two refinements from lists to ordered lists and vectors to get a refinement from lists to ordered vectors whose promotion predicate is the pointwise conjunction of the promotion predicates of the two refinements. This is easy for the externalist side of the refinement, but for the internalist side, we need to derive the datatype of ordered vectors from the datatypes of ordered lists and vectors, which is not possible unless we can tap more deeply into the structure of datatypes and manipulate such structure — that is, we need to do **datatype-generic programming** (Section 2.4). Hence enter ornaments. With ornaments, we can express intensional relationship between datatype declarations, which can be exploited for deriving composite datatypes like ordered vectors. This intensional relationship is easy to establish and induces refinement families (Section 3.3), so the difficulty of preparing refinement families is also dramatically reduced.

3.2 Ornaments

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

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

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

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

(Implicit arguments can be explicitly supplied in curly braces.) Our goal in this section is to construct a universe for such horizontal natural transformations between the base functors arising as decodings of descriptions. The inhabitants of this universe are called **ornaments**. By encoding the relationship between datatype descriptions as a universe, whose inhabitants are analysable syntactic objects, we will not only be able to derive conversion functions between datatypes, but even compute new datatypes that are related to old ones in prescribed ways (e.g., by parallel composition in Section 3.2.3), which is something we cannot achieve if we simply write the conversion functions directly.

3.2.1 Universe construction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$
\begin{array}{llcll}
erase\ (\mathsf{v}\ [\,]) & \blacksquare & = & \blacksquare & \\
erase\ (\mathsf{v}\ (\mathsf{refl} :: eqs)) & (x\ ,\ xs) & = & x\ ,\ erase\ (\mathsf{v}\ eqs)\ xs & \text{-- } x \text{ retained} \\
erase\ (\sigma\ S\ O) & (s\ ,\ xs) & = & s\ ,\ erase\ (O\ s)\ \ xs & \text{-- } s \text{ copied} \\
erase\ (\Delta\ T\ O) & (t\ ,\ xs) & = & erase\ (O\ t)\ \ xs & \text{-- } t \text{ discarded} \\
erase\ (\nabla\ s\ O) & xs & = & s\ ,\ erase\ O\ \ xs & \text{-- } s \text{ inserted}
\end{array}
$$

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 $map\ e\ js \equiv is$ (where map is the usual functorial map on lists). To make it easier to analyse a proof of $map\ e\ js \equiv is$ 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

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

$$\{js : \mathsf{List}\ J\}\ \{is : \mathsf{List}\ I\}\ (eqs : \mathbb{E}\ e\ js\ is) \rightarrow \mathbb{E}\ e\ (j :: js)\ (i :: is)$$

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

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

- The Δ case of ROrn states that σ 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 σ 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:

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

$$; \text{'cons} \;\mapsto\; \Delta[\,\_ : A\,]\;\; \mathsf{v}\;(\mathsf{refl} :: [\,])\;\}$$

where the erasure function '!' is $\lambda\,\_\;\mapsto\;\blacksquare$. There is only one response ornament in *NatD-ListD A* since the datatype of lists is trivially indexed. The constructor tag is preserved (σ ListTag), and in the cons case, the list element field is marked as additional by Δ. Consequently, the forgetful function

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

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 $A\;:\;\mathsf{Set}$, there is an ornament from the description *ListD A* of lists to the description *VecD A* of vectors:

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

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

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

computes the underlying list of a vector. □

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.

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

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

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

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

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

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

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

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

Figure 3.1 Definitions for ornamental descriptions.

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

Ornamental descriptions are designed for this purpose. The related definitions are shown in Figure 3.1 and closely follow the definitions for ornaments, having a upper-level type $\mathsf{OrnDesc}$ of ornamental descriptions which refers to a lower-level datatype $\mathsf{ROrnDesc}$ of response ornamental descriptions. An ornamental description looks like an annotated description, on which we can use a greater variety of constructors to mark differences from the template description. We think of an ornamental description

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

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

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

and the ornament is

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

Example (*ordered lists as an ornamentation of lists*). We can define ordered lists by an ornamental description, using the description of lists as the template:

$$OrdListOD \; : \; \mathsf{OrnDesc}\; Val \; ! \; (ListD\; Val)$$
$$OrdListOD\; (\mathsf{ok}\; b) \; =$$

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

If we read *OrdListOD* as an annotated description, we can think of the *leq* field as being marked as additional (relative to the description of lists) by using Δ rather than σ. We write

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

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

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

to get an ornament from lists to ordered lists. □

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

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

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

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

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

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

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

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

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

where

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

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

Remark (*index-first universes*)**.** The datatype of response ornamental descriptions is a good candidate for receiving an index-first reformulation. Since the structure of a response ornamental description is guided by the template response description, ROrnDesc is much more clearly presented in the index-first style:

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

If the template response description is v *is*, then we can specify a list of indices refining *is* (by v); if it is σ *S D*, then we can either copy (σ) or delete (∇) the field; finally, whatever the template is, we can always choose to create (Δ) a new field. This thesis maintains a separation between Agda datatypes and index-first datatypes, in particular using the former to construct a universe for

$$
\begin{array}{l}
\textbf{record}\ \_{\bowtie}\_\ \{I\ J\ K\ :\ \mathsf{Set}\}\ (e\ :\ J \to I)\ (f\ :\ K \to I)\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ \_,\_ \\
\quad \textbf{field} \\
\qquad \{i\}\ :\ I \quad \text{-- implicit field} \\
\qquad j \quad\ :\ e^{-1}\ i \\
\qquad k \quad\ :\ f^{-1}\ i
\end{array}
$$

$$
\begin{array}{l}
\mathit{pull}\ :\ \{I\ J\ K\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{f\ :\ K \to I\} \to e \bowtie f \to I \\
\mathit{pull}\ =\ \_{\bowtie}\_.i
\end{array}
$$

$$
\begin{array}{l}
\mathit{outl}_{\bowtie}\ :\ \{I\ J\ K\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{f\ :\ K \to I\} \to e \bowtie f \to J \\
\mathit{outl}_{\bowtie}\ =\ \mathit{und} \circ \_{\bowtie}\_.j
\end{array}
$$

$$
\begin{array}{l}
\mathit{outr}_{\bowtie}\ :\ \{I\ J\ K\ :\ \mathsf{Set}\}\ \{e\ :\ J \to I\}\ \{f\ :\ K \to I\} \to e \bowtie f \to K \\
\mathit{outr}_{\bowtie}\ =\ \mathit{und} \circ \_{\bowtie}\_.k
\end{array}
$$

Figure 3.2 Definitions for set-theoretic pullbacks.

the latter, but it is conceivable that ornaments and ornamental descriptions can be incorporated into a type theory with self-encoding index-first universes like the one presented by Chapman et al. [2010]. □

3.2.3 Parallel composition of ornaments

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

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

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

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

where $e \bowtie f$ is the set-theoretic pullback of $e : J \to I$ and $f : K \to I$, i.e., it is isomorphic to $\Sigma[\,jk : J \times K\,]\ \ e\ (outl\ jk)\ \equiv\ f\ (outr\ jk)$; related definitions are shown in Figure 3.2. Intuitively, since both O and P encode modifications to the same base description D, we can commit all modifications encoded by O and P to D to get a new description $\lfloor O \otimes P \rfloor$, and encode all these modifications in one ornament $\lceil O \otimes P \rceil$. The forgetful function of the ornament $\lceil O \otimes P \rceil$ removes all modifications, taking $\mu\ \lfloor O \otimes P \rfloor$ all the way back to the base datatype $\mu\ D$; there are also two **difference ornaments**

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

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

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

are extensionally equal to $forget\ \lceil O \otimes P \rceil$. (The diagrams foreshadow our characterisation of parallel composition as a category-theoretic pullback in Chapter 4; we will make their meanings precise there.)

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

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

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

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

and an ornament $\lceil \mathit{OrdVecOD} \rceil$ whose forgetful function converts ordered vectors to plain lists, retaining the list elements. The forgetful functions of the difference ornaments convert ordered vectors to ordered lists and vectors, removing only length and ordering information respectively. □

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

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

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

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

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

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

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

$$
\begin{array}{l}
\mathit{fromEq} \,:\, \{I\,J \,:\, \mathsf{Set}\}\ (e \,:\, J \to I)\ \{j \,:\, J\}\ \{i \,:\, I\} \to e\ j \equiv i \to e^{\,-1}\ i \\
\mathit{fromEq}\ e\ \{j\}\ \mathsf{refl} \;=\; \mathsf{ok}\ j
\end{array}
$$

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

mutual

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

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

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

Figure 3.3 Definitions for parallel composition of ornaments.

- 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 $pcROD\text{-}double\nabla$ puts the deletion into the resulting response ornamental description after matching the proof of the equality field with refl.

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

  (The implicit argument $\{s := s\}$ is the one named s in the type of $pcROD\text{-}double\nabla$ — the 's' to the left of ':=' is the name of the argument, while the 's' to the right is a pattern variable. This syntax allows us to skip the nine implicit arguments before this one.) It might seem bizarre that two deletions results in a new field (and a deletion), but consider this informally described scenario: A field $\sigma\ S$ in the base response description is refined by two independent response ornaments

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

  and

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

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

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

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

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

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

where $\boxed{\text{lighter box}}$ indicates modifications from $\lceil\, \mathit{OrdListOD} \,\rceil$ and $\boxed{\text{darker box}}$ from *ListD-VecD Val*. □

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

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

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

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

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

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

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

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

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

mutual

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

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

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

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

Figure 3.4 Definitions for left difference ornament.

3.3 Refinement semantics of ornaments

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

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

which is called the **refinement semantics** of ornaments. The construction of *RSem* begins in Section 3.3.1 and continues into Chapter 4 (where we introduce a lightweight categorical organisation). Another important aspect of the translation is from parallel composition of ornaments to refinements whose promotion predicate is pointwise conjunctive. This begins in Section 3.3.2 and also continues into Chapter 4.

3.3.1 Optimised predicates

Our task in this section is to construct a promotion predicate

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

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

$$\begin{aligned}
\mathit{two} \;=\; & \mathsf{con}\ (\text{‘}\mathsf{cons}\ , \\
& \mathsf{con}\ (\text{‘}\mathsf{cons}\ ,
\end{aligned}$$

$$\begin{array}{l} \quad\mathsf{con}\ (\text{‘}\mathsf{nil}\quad , \\ \qquad \blacksquare)\ ,\ \blacksquare)\ ,\ \blacksquare)\ :\ \mu\ \mathit{NatD}\ \blacksquare \end{array}$$

to a list, an optimised promotion proof would look like

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

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

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

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 manner, from the inhabitant being promoted, which appears in the index of the type of a proof. The deletion is independent of O and can be performed by the singleton ornament for D (Section 3.2.2), so the desired ornamental description is produced by the parallel composition of O and $\lceil \mathit{singletonOD}\ D \rceil$:

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

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

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

$$\mathsf{OptP}\ O\ \{i\}\ j\ x \ = \ \mu\ \lfloor OptPOD\ O \rfloor\ (j\ ,(\mathsf{ok}\ (i\ ,x)))$$

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

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

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

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

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

where lighter box indicates contributions from $\lceil OrdListOD \rceil$ and darker box from $\lceil singletonOD\ (ListD\ Val) \rceil$. Since a proof of $\mathsf{Ordered}\ b\ xs$ consists of exactly the inequality proofs necessary for ensuring that *xs* is ordered and bounded below by *b*, its representation is optimised, justifying the name "optimised predicate". □

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

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

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

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

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

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

indexfirst data Length : Nat $\to$ List A $\to$ Set **where**
  Length zero [] $\ni$ nil
  Length zero $(a :: as)$ $\not\ni$
  Length (suc n) [] $\not\ni$
  Length (suc n) $(a :: as)$ $\ni$ cons (l : Length n as)

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

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

RSem : $\{I\,J : \mathsf{Set}\}\;\{e : J \to I\}\;\{D : \mathsf{Desc}\;I\}\;\{E : \mathsf{Desc}\;J\} \to$
  Orn $e\;D\;E \to$ FRefinement $e\;(\mu\;D)\;(\mu\;E)$
RSem $O\;j$ = **record** { P = OptP $O\;j$
  ; i = *ornConvIso* $O\;j$ }

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

ornConvIso :

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

The construction of *ornConvIso* is deferred to 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 $\lceil \lceil \mathit{OrdListOD} \rceil \otimes \mathit{ListD\text{-}VecD}\ \mathit{Val} \rceil$ from lists to ordered vectors is

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

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

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

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

Along with the ornamental conversion isomorphisms, the construction of the modularity isomorphisms is deferred to 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 {A B : Set} (r : Refinement A B) : Set₁ where
  field
    P : A → Set
    i : (a : A) → Refinement.P r a ≅ P a
```

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 : {A B : Set} {r : Refinement A B} → Swap r → Refinement A B
toRefinement s = record { P = Swap.P s
                        ; i = { }₀ }
```

where Goal 0 is the new conversion isomorphism

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

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

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

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

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

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

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

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

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

where

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

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

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

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

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

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

as the promotion predicate, where

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

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

3.4 Examples

To demonstrate the use of the ornament–refinement framework, in Section 3.4.1 we first conclude the example about insertion into a list introduced in Section 2.5, and then we look at two dependently typed heap data structures adapted from Okasaki's work on purely functional data structures [1999]. Of the latter two examples,

- the first one about **binomial heaps** (Section 3.4.2) shows that Okasaki's idea of **numerical representations** can be elegantly captured by ornaments and the coherence properties computed with upgrades, and
- the second one about **leftist heaps** (Section 3.4.3) demonstrates the power of parallel composition of ornaments by treating heap ordering and leftist balancing properties modularly.

$$
\begin{array}{l}
\quad \text{-- the upgraded function type has an extra argument} \\
\mathit{new} : \{A : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\} \to \\
\qquad ((i : I) \to \mathsf{Upgrade}\ A\ (X\ i)) \to \mathsf{Upgrade}\ A\ ((i : I) \to X\ i) \\
\mathit{new}\ I\ u \ = \ \textbf{record}\ \{\ P \ = \ \lambda\, a \mapsto (i : I) \to \mathsf{Upgrade}.P\ (u\ i)\ a \\
\qquad\qquad\qquad\qquad ;\ C \ = \ \lambda\, a\ x \mapsto (i : I) \to \mathsf{Upgrade}.C\ (u\ i)\ a\ (x\ i) \\
\qquad\qquad\qquad\qquad ;\ u \ = \ \lambda\, a\ p\ i \mapsto \mathsf{Upgrade}.u\ (u\ i)\ a\ (p\ i) \\
\qquad\qquad\qquad\qquad ;\ c \ = \ \lambda\, a\ p\ i \mapsto \mathsf{Upgrade}.c\ \ (u\ i)\ a\ (p\ i)\ \} \\
\textbf{syntax}\ \mathit{new}\ I\ (\lambda\, i \mapsto u) \ = \ \forall^{+}[\, i : I\,]\ \ u \\
\quad \text{-- implicit version of } \mathit{new} \\
\mathit{new}' : \{A : \mathsf{Set}\}\ (I : \mathsf{Set})\ \{X : I \to \mathsf{Set}\} \to \\
\qquad ((i : I) \to \mathsf{Upgrade}\ A\ (X\ i)) \to \mathsf{Upgrade}\ A\ (\{i : I\} \to X\ i) \\
\mathit{new}'\ I\ u \ = \ \textbf{record}\ \{\ P \ = \ \lambda\, a \mapsto \{i : I\} \to \mathsf{Upgrade}.P\ (u\ i)\ a \\
\qquad\qquad\qquad\qquad ;\ C \ = \ \lambda\, a\ x \mapsto \{i : I\} \to \mathsf{Upgrade}.C\ (u\ i)\ a\ (x\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ u \ = \ \lambda\, a\ p\ \{i\} \mapsto \mathsf{Upgrade}.u\ (u\ i)\ a\ (p\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ c \ = \ \lambda\, a\ p\ \{i\} \mapsto \mathsf{Upgrade}.c\ \ (u\ i)\ a\ (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 ((i : I) \to \mathsf{Upgrade}\ (X\ i)\ (Y\ i)) \to \mathsf{Upgrade}\ ((i : I) \to X\ i)\ ((i : I) \to Y\ i) \\
\mathit{fixed}\ I\ u \ = \ \textbf{record}\ \{\ P \ = \ \lambda f \mapsto (i : I) \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ i) \\
\qquad\qquad\qquad\qquad ;\ C \ = \ \lambda f\ g \mapsto (i : 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 ((i : I) \to \mathsf{Upgrade}\ (X\ i)\ (Y\ i)) \to \mathsf{Upgrade}\ (\{i : I\} \to X\ i)\ (\{i : I\} \to Y\ i) \\
\mathit{fixed}'\ I\ u \ = \ \textbf{record}\ \{\ \ P \ = \ \lambda f \mapsto \{i : I\} \to \mathsf{Upgrade}.P\ (u\ i)\ (f\ \{i\}) \\
\qquad\qquad\qquad\qquad ;\ \ C \ = \ \lambda f\ g \mapsto \{i : 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 3.5 More combinators on upgrades.

3.4.1 Insertion into a list

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

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

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

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

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

$$insert_V \;:\; Val \to \{n : \mathsf{Nat}\} \to \mathsf{Vec}\; Val\; n \to \mathsf{Vec}\; Val\; (\mathsf{suc}\; n)$$

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

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

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

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

which satisfies the coherence property

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

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

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

is manufactured with the help of the upgrade

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

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

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

and is inhabited by

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

which is strikingly similar to $\mathit{insert_{EOV}}$ in Section 2.5.

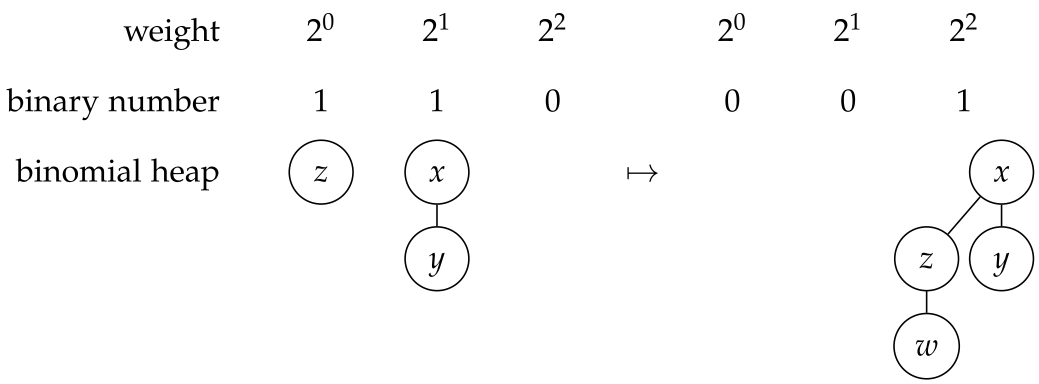

Figure 3.6 **Left:** a binomial heap of size 3 consisting of two binomial trees storing elements x, y, and z. **Right:** a possible result of inserting an element w into the heap. (Note that the digits of the underlying binary numbers are ordered with the least significant digit first.)

3.4.2 Binomial heaps

We are all familiar with the idea of **positional number systems**, in which we represent numbers as a list of digits. Each position in a list of digits is associated with a weight, and the interpretation of the list is the weighted sum of the digits. (For example, the weights used for binary numbers are powers of 2.) Some container data structures and associated operations strongly resemble positional representations of natural numbers and associated operations. For example, a **binomial heap** (illustrated in Figure 3.6) can be thought of as a binary number in which every 1-digit stores a **binomial tree** — the actual place for storing elements — whose size is exactly the weight of the digit. The number of elements stored in a binomial heap is therefore exactly the value of the 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

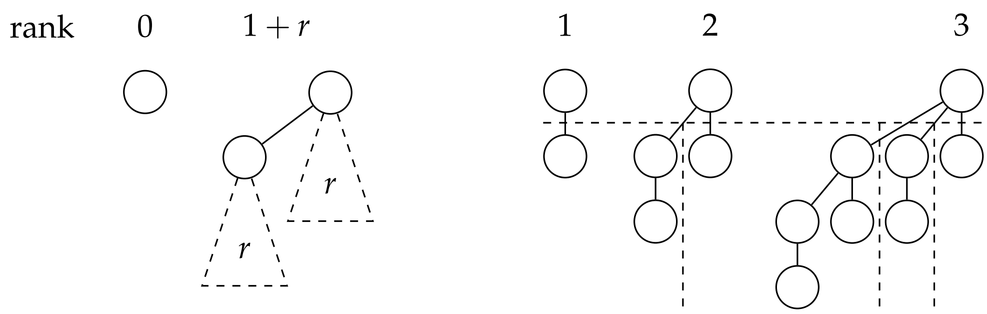

Figure 3.7 **Left:** inductive definition of binomial trees. **Right:** decomposition of binomial trees of ranks 1 to 3.

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

Binomial trees

The basic building blocks of binomial heaps are **binomial trees**, in which elements are stored. Binomial trees are defined inductively on their **rank**, which is a natural number (see Figure 3.7):

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

From this definition we can readily deduce that a binomial tree of rank r has 2^r elements. To actually define binomial trees as a datatype, however, an alternative view is more useful: a binomial tree of rank r is constructed by attaching binomial trees of ranks 0 to $r - 1$ under a root node. (Figure 3.7 shows how binomial trees of ranks 1 to 3 can be decomposed according to this view.) We thus define the datatype $\mathsf{BTree} : \mathsf{Nat} \to \mathsf{Set}$ — which is indexed with the rank of binomial trees — as follows: for any rank $r : \mathsf{Nat}$, the type $\mathsf{BTree}\ r$ has a field of type *Val* — which is the root node — and r recursive positions indexed from $r - 1$ down to 0. This is directly encoded as a description:

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

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

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 $\mathsf{BTree}\ r$ is of the form $\mathsf{con}\ (x\ , ts)$ where $x : \mathit{Val}$ is the root element and $ts : \mathbb{P}\ (\mathit{descend}\ r)\ \mathsf{BTree}$ is a series of sub-trees.

The most important operation on binomial trees is combining two smaller binomial trees of the same rank into a larger one, which corresponds to carrying in positional arithmetic. Given two binomial trees of the same rank r, one can be *attach*ed under the root of the other, forming a single binomial tree of rank $1 + r$ — this is exactly the inductive definition of binomial trees.

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

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

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

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:

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

The intended interpretation of binary numbers is given by

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

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 $\mathsf{BHeap} : \mathsf{Nat} \rightarrow \mathsf{Set}$ is indexed with a "starting rank": if a binomial heap of type $\mathsf{BHeap}\ r$ is nonempty (i.e., not nil), then its first digit has rank r (and stores a binomial tree of rank r when the digit is one), and the rest of the heap is indexed with $1 + r$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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*. Insertion can now be implemented as

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

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:

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

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*:

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

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

3.4.3 Leftist heaps

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

> Leftist heaps [...] are heap-ordered binary trees that satisfy the **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).

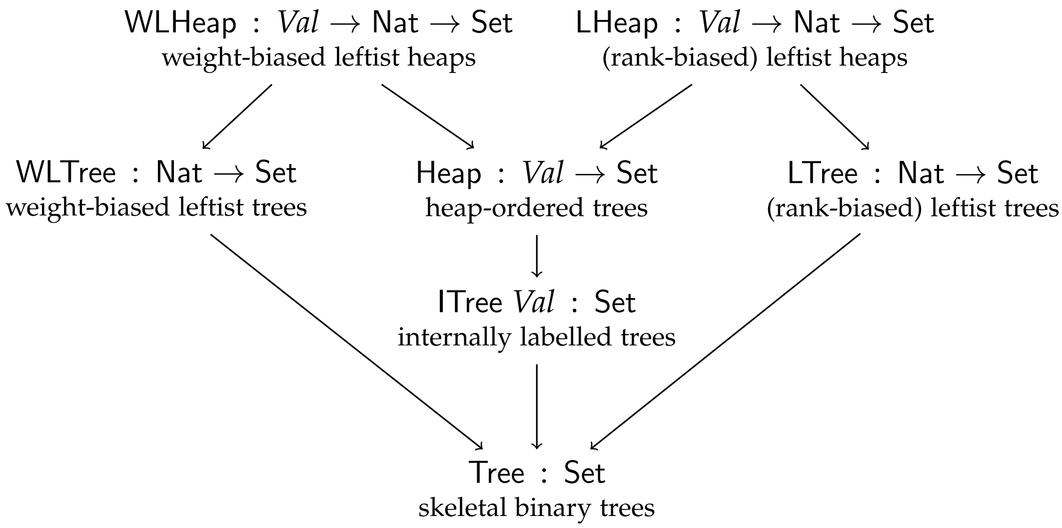

Figure 3.8 Datatypes about leftist heaps and their ornamental relationships.

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 3.8.

Datatypes leading to leftist heaps

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

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

TreeD : Desc ⊤
TreeD • = σ TreeTag λ { 'nil   ↦ v []
```

$$; \text{'node} \mapsto \mathsf{v}\,(\blacksquare :: \blacksquare :: [\,])\ \}$$

indexfirst data Tree : Set **where**

$$\begin{aligned}\mathsf{Tree} \ni\ & \mathsf{nil} \\ |\ & \mathsf{node}\ (t : \mathsf{Tree})\ (u : \mathsf{Tree})\end{aligned}$$

Leftist trees — skeletal binary trees satisfying the leftist property — are then an ornamented version of Tree. The datatype LTree : Nat → Set of leftist trees is indexed with the rank of the root of the trees. The constructor choices can be determined from the rank: the only node that can have rank zero is the empty node nil; otherwise, when the rank of a node is non-zero, it must be an internal node constructed by the node constructor, which enforces the leftist property. (Below we overload $\_\leqslant\_$ to also denote the decidable total ordering on Nat.)

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

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

$$\begin{aligned}&\mathsf{Tree\ zero} \ni \mathsf{nil} \\ &\mathsf{Tree}\ (\mathsf{suc}\ r) \ni \mathsf{node}\ \{l : \mathsf{Nat}\}\ (r{\leqslant}l : r \leqslant l)\ (t : \mathsf{Tree}\ l)\ (u : \mathsf{Tree}\ r)\end{aligned}$$

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

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

indexfirst data Heap : *Val* → Set **where**

$$\begin{aligned}\mathsf{Heap}\ b \ni\ & \mathsf{nil} \\ |\ & \mathsf{node}\ (x : \mathit{Val})\ (b{\leqslant}x : b \leqslant x)\ (t : \mathsf{Heap}\ x)\ (u : \mathsf{Heap}\ x)\end{aligned}$$

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

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

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

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

$$
\begin{aligned}
&\mathsf{Heap}' : \mathit{Val} \to \mathsf{Set}\\
&\mathsf{Heap}'\ b \;=\; \mathsf{OptP}\ \lceil \mathit{HeapOD} \rceil\ (\mathsf{ok}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{Heap}' : \mathit{Val} \to \mathsf{Tree} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{Heap}'\ b\ \mathsf{nil} \qquad\qquad \ni\ \mathsf{nil}\\
&\quad \mathsf{Heap}'\ b\ (\mathsf{node}\ t\ u)\ \ni\ \mathsf{node}\ (x : \mathit{Val})\ (b{\leqslant}x : b \leqslant x)\\
&\qquad\qquad\qquad\qquad\qquad (t' : \mathsf{Heap}\ x\ t)\ (u' : \mathsf{Heap}\ x\ u)
\end{aligned}
$$

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

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

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:

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

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

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

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

on which we can do preorder traversal:

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

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

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

where *idRef* is the identity refinement:

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

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

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

(Sequential composition of ornaments will be introduced in Chapter 4.) Specialising *preorder* by the ornament gives preorder traversal of a leftist heap.

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

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

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

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

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

Figure 3.9 Merging two leftist heaps. Proof terms about ordering are coloured grey to aid comprehension (taking inspiration from — but not really employing — Bernardy and Guilhem's "type theory in color" [2013]).

with which we can easily implement insertion of a new element (by merging with a singleton heap) and deletion of the minimum element (by deleting the root and merging the two sub-heaps). The definition of *merge* is shown in Figure 3.9. It is a more precisely typed version of Okasaki's implementation, split into two mutually recursive functions to make it clear to Agda's termination checker that we are doing two-level induction on the ranks of the two input heaps. When one of the ranks is zero, meaning that the corresponding heap is nil, we simply return the other heap (whose bound is suitably relaxed) as the result. When both ranks are nonzero, meaning that both heaps are nonempty, we compare the roots of the two heaps and recursively merge the heap with the larger root into the right branch of the heap with the smaller root. The recursion is structural because the rank of the right branch of a nonempty heap is strictly smaller. There is a catch, however: the rank of the new right sub-heap resulting from the recursive merging might be larger than that of the left sub-heap, violating the leftist property, so there is a helper function *makeT* that swaps the sub-heaps when necessary.

Weight-biased leftist heaps

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

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

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

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

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

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

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

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

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

3.5 Discussion

Ornaments were first proposed by McBride [2011]. This thesis defines ornaments as relations between descriptions (indexed with an erasure function), and rebrands McBride's ornaments as ornamental descriptions. One obvious advantage of relational ornaments is that they can arise between existing descriptions, whereas ornamental descriptions always produce new descriptions at the more informative end. This makes it possible to complete the commutative square of parallel composition with difference ornaments. Another consequence is that there can be multiple ornaments between a pair of descriptions. For example, consider the following description of a datatype consisting of two fields of the same type:

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

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

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

and the "swapping" ornament

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

whose forgetful function swaps the two fields. The other advantage of relational ornaments is that they allow new datatypes to arise at the less informative end. For example, **coproduct of signatures** as used in, e.g., data types à la carte [Swierstra, 2008], can be implemented naturally with relational ornaments but not with ornamental descriptions. Below we sketch a simplistic implementation: Consider (a simplistic version of) **tagged descriptions** [Chapman et al., 2010], which are descriptions that, for any index request, always respond with a constructor field first. A tagged description indexed by $I : \mathsf{Set}$

thus consists of a family of types $C : I \rightarrow \mathsf{Set}$, 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.

$$\mathsf{TDesc} : \mathsf{Set} \rightarrow \mathsf{Set}_1$$
$$\mathsf{TDesc}\ I \ = \ \Sigma[\, C : I \rightarrow \mathsf{Set}\,]\ ((i : I) \rightarrow C\ i \rightarrow \mathsf{RDesc}\ I)$$

Tagged descriptions are decoded to ordinary descriptions by

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

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

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

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

$$\mathbf{data}\ \_+\_\ (A\ B : \mathsf{Set}) : \mathsf{Set}\ \mathbf{where}$$
$$\quad \mathsf{inl} : A \rightarrow A + B$$
$$\quad \mathsf{inr} : B \rightarrow A + B$$

$$\_\triangledown\_ : \{A\ B\ C : \mathsf{Set}\}\ (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow A + B \rightarrow C$$
$$(f \triangledown g)\ (\mathsf{inl}\ a) \ = \ f\ a$$
$$(f \triangledown g)\ (\mathsf{inr}\ b) \ = \ g\ b$$

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 $\lfloor tD' \rfloor_T$:

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

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

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

Dagand and McBride [2012b] later adapted McBride's original ornaments to index-first datatypes, and also proposed "reornaments" as a more efficient representation of promotion predicates, taking full advantage of index-first datatypes. Reornaments are reimplemented in this thesis as optimised predicates using parallel composition, as a result of which we can derive properties about optimised predicates using pullback properties of parallel composition in Chapter 4. Dagand and McBride also extended the notion of ornaments to "functional ornaments", which we generalise to refinements and upgrades. The refinement–upgrade approach is logically clearer and more flexible as it allows us to decouple two constructions:

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

Also, compared to functional ornaments, which are formulated syntactically as a universe and then interpreted to types and operations, upgrades skip syntactic formulation and simply bundle relevant types and operations together, which are then composed semantically by the upgrade combinators. The upgrade mechanism can thus be more easily extended by defining new combinators (which we actually do in Section 5.3.1). In contrast, had we defined upgrades as a universe, we would have had to employ more complex techniques like data types à la carte [Swierstra, 2008] to gain extensibility. The complexity would not have been justified, because constructing a universe for upgrades in their present form offers no benefit: A universe is helpful only when it is necessary to determine the range of syntactic forms, either for nontrivial computation on the syntactic forms or for facilitation of defining new interpretations of the syntactic forms. Neither is the case with upgrades: we do not need to manipulate the syntactic forms of upgrades, nor do we need to obtain semantic entities other than those captured by the fields of Upgrade. In contrast, ornaments do need a universe: we need to know all possible syntactic forms of ornaments in order to compose them in parallel, which cannot be done if all we have are the optimised predicates and ornamental conversion isomorphisms, i.e., the refinement semantics. Indeed, this was what prompted us to go from refinements to ornaments, right before Section 3.2. The universe of ornaments might appear complex, but the complexity is justified by, in particular, the ability to compose ornaments in parallel.

The idea of viewing vectors as promotion predicates was first proposed by Bernardy [2011, page 82], and is later generalised to "type theory in color" [Bernardy and Guilhem, 2013], which uses modalities inspired by colors in typing to manage relative irrelevance of terms and erasure of irrelevant terms. For simple applications like the ones offered in Section 3.4, type theory in color and ornamentation offer similar approaches, with the former providing more native support for erasure of terms and derivation of promotion predicates. Ornaments, however, are fully computational due to the presence of deletion (∇), which allows arbitrary computations, and can thus specify relationship between datatypes beyond erasure. (Chapter 6 will offer a clearer view on the computational power of ornaments.)

It is worth noting that

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

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

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

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 (Section 3.2.3) has a sensible definition, 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

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

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

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 ornamentation (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 ornamentation.

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). The categories and functors used in this chapter are summarised in Figure 4.1.
- Parallel composition merges two compatible ornaments and does nothing more; in other words, it computes the least informative ornament that con-

tains 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 be a categorical **pullback** (Section 4.2).

- Universal constructions are unique up to isomorphism, so it is convenient to establish isomorphisms about universal constructions. The status of parallel composition being a pullback can thus help to construct the ornamental conversion isomorphisms (in Section 4.3.1) and the modularity isomorphisms (in Section 4.3.2).

Section 4.4 concludes with some discussion.

4.1 Categories and functors

We define the general notions of categories and functors in Section 4.1.1 and then concrete categories and functors specifically about ornaments and refinements in Section 4.1.2. Categories and functors by themselves are uninteresting, though; it is the purely categorical structures defined on top of categories and functors that make the categorical language worthwhile. We introduce the first such definition — categorical isomorphisms – in Section 4.1.3, and more in Section 4.2.

4.1.1 Basic definitions

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 $\mathbb{F}\text{UN}$ of (small) sets and (total) functions is

$$
\begin{aligned}
&\mathbb{F}\text{UN}\text{-}\mathit{graph} \;:\; \mathsf{Graph}\\
&\mathbb{F}\text{UN}\text{-}\mathit{graph} \;=\; \mathbf{record}\;\{\; \mathit{Object} \;=\; \mathsf{Set}\\
&\qquad\qquad\qquad\qquad\quad ;\; \_{\Rightarrow}\_ \;=\; \lambda\, A\; B \mapsto A \to B \;\}
\end{aligned}
$$

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

$$\_\cdot\_ \;:\; \{X\;Y\;Z \;:\; \mathit{Object}\} \to (Y \Rightarrow Z) \to (X \Rightarrow Y) \to (X \Rightarrow Z)$$

which has left and right 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 $\mathbb{F}\text{UN}$-*graph* is implicitly instantiated as Graph $\{1\}$ $\{0\}$, since both Set is of type Set_1 and and any $A \to B$ (where A, B : Set) are of type Set (= Set_0), and Graph $\{1\}$ $\{0\}$ itself is of type Set_2, whose level is computed by taking the successor of the maximum of the two level arguments.) 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 $\mathbb{F}\text{UN}$, for example, it is necessary to identify functions up to extensional equality (so uniqueness of morphisms in universal properties would make sense). As stated in Section 2.3, such equalities need to be explicitly managed in AGDA's intensional setting, and one way is to use **setoids** [Barthe et al., 2003] — sets with an explicitly specified equivalence relation — to represent sets of morphisms. Subsequently, functions defined between setoids need to be proved to respect the equivalences. 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:

$$\mathbf{record}\;\mathsf{Setoid}\;\{c\;d \;:\; \mathsf{Level}\} \;:\; \mathsf{Set}\;(\mathsf{suc}\;(c \sqcup d))\;\mathbf{where}$$

$$
\begin{array}{l}
\textbf{field} \\
\quad \mathit{Carrier} \;:\; \mathsf{Set}\; c \\
\quad \_{\approx}\_ \quad\;\, : \mathit{Carrier} \to \mathit{Carrier} \to \mathsf{Set}\; d \\
\quad \mathit{refl} \quad\; : \{x \,:\, \mathit{Carrier}\} \to x \approx x \\
\quad \mathit{sym} \quad\, : \{x\; y \,:\, \mathit{Carrier}\} \to x \approx y \to y \approx x \\
\quad \mathit{trans} \;\, : \{x\; y\; z \,:\, \mathit{Carrier}\} \to x \approx y \to y \approx z \to x \approx z
\end{array}
$$

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

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

Proofs of the three laws are omitted from the presentation.

The type of categories is then defined as a record containing a set of objects, a collection of setoids of morphisms indexed by source and target 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, e.g., the AoPA library [Mu et al., 2009]).

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

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

$$
\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}\ \ \ :\ \{X\ :\ \mathit{Object}\} \to (X \Rightarrow X) \\
\qquad \mathit{id\text{-}l}\ \ \ :\ \{X\ Y\ :\ \mathit{Object}\}\ (f\ :\ X \Rightarrow Y) \to \mathit{id} \cdot f\ \approx\ f \\
\qquad \mathit{id\text{-}r}\ \ \ :\ \{X\ Y\ :\ \mathit{Object}\}\ (f\ :\ X \Rightarrow Y) \to 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 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 f\ \approx\ g \to h \cdot f\ \approx\ h \cdot g \\
\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} \\
\qquad \mathit{object} \qquad\ :\ \mathit{Object}\ C \to \mathit{Object}\ D \\
\qquad \mathit{morphism}\ \ :\ \{X\ Y\ :\ \mathit{Object}\ C\} \to X \Rightarrow_C Y \to \mathit{object}\ X \Rightarrow_D \mathit{object}\ Y \\
\qquad \mathit{equiv\text{-}preserving}\ :\ \{X\ Y\ :\ \mathit{Object}\ C\}\ \{f\ g\ :\ X \Rightarrow_C Y\} \to \\
\qquad\qquad\qquad\qquad\quad f\ \approx_C\ g \to \mathit{morphism}\ f\ \approx_D\ \mathit{morphism}\ g \\
\qquad \mathit{id\text{-}preserving} \quad\ :\ \{X\ :\ \mathit{Object}\ C\} \to \mathit{morphism}\ (\mathit{id}\ C\ \{X\})\ \approx_D\ \mathit{id}\ D\ \{\mathit{object}\ X\} \\
\qquad \mathit{comp\text{-}preserving}\ :\ \{X\ Y\ Z\ :\ \mathit{Object}\ C\}\ (f\ :\ Y \Rightarrow_C Z)\ (g\ :\ X \Rightarrow_C Y) \to \\
\qquad\qquad\qquad\qquad\quad \mathit{morphism}\ (f \cdot_C g)\ \approx_D\ (\mathit{morphism}\ f \cdot_D \mathit{morphism}\ g)
\end{array}
$$

Figure 4.2 Definitions of categories and functors. Subscripts are used to indicate to which category an operator belongs.

```
FAM : Category
FAM = record
  { Object    = Σ[ I : Set ] I → Set
  ; Morphism  = λ { (J , Y) (I , X) ↦ record
                  { Carrier = Σ[ e : J → I ] Y ⇉ (X ∘ e)
                  ; _≈_     = λ { (e , u) (e′ , u′) ↦
                                  (e ≐ e′) × ((j : J) → u {j} ≋ u′ {j}) }
                  ; proofs of laws } }
  ; _·_ = λ { (e , u) (f , v) ↦ e ∘ f , (λ {k} ↦ u {f k} ∘ v {k}) }
  ; id  = (λ x ↦ x) , (λ {i} x ↦ x)
  ; proofs of laws }

FREF : Category
FREF = record
  { Object    = Σ[ I : Set ] I → Set
  ; Morphism  = λ { (J , Y) (I , X) ↦ record
                  { Carrier = Σ[ e : J → I ] FRefinement e X Y
                  ; _≈_     = λ { (e , rs) (e′ , rs′) ↦
                                  (e ≐ e′) ×
                                  ((j : J) → Refinement.forget (rs (ok j)) ≋
                                             Refinement.forget (rs′ (ok j))) }
                  ; proofs of laws } }
  ; proofs of laws }

ORN : Category
ORN = record
  { Object    = Σ[ I : Set ] Desc I
  ; Morphism  = λ { (J , E) (I , D) ↦ record
                  { Carrier = Σ[ e : J → I ] Orn e D E
                  ; _≈_     = λ { (e , O) (f , P) ↦ OrnEq O P }
                  ; proofs of laws } }
  ; _·_ = λ { (e , O) (f , P) ↦ e ∘ f , O ⊙ P }
  ; id  = λ { {I , D} ↦ (λ i ↦ i) , idOrn D }
  ; proofs of laws }
```

Figure 4.3 (Partial) definitions of the categories FAM, FREF, and ORN.

Another important category that we will make use of is $\mathbb{F}\textsc{am}$ (Figure 4.3), 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 $j\;:\;J$. Morphism composition is componentwise composition, and morphism equivalence is defined to be componentwise extensional equality. (The morphism equivalence is formulated with the help of McBride's "John Major" heterogeneous equality $\_{\approxeq}\_$ [McBride, 1999] — the equivalence $\_\dot{\approxeq}\_$ is pointwise heterogeneous equality — since 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.)

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 4.2: 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 $\mathbb{F}\textsc{am}$ to $\mathbb{F}\textsc{un}$, one summing components together

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

and the other extracting the index part.

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

Proofs of the functor laws are omitted from the presentation.

4.1.2 Categories and functors for refinements and ornaments

Some constructions in Chapter 3 can now be organised under several categories (shown in Figure 4.3) and functors. For a start, we already saw that refinements are interesting only because of their intensional contents; extensionally they amount only to their forgetful functions. This is reflected in an isomorphism of categories between the category $\mathbb{F}\textsc{am}$ and the category $\mathbb{F}\textsc{ref}$ of type families and refinement families (i.e., there are two functors back and forth inverse to each other). An object in $\mathbb{F}\textsc{ref}$ is an indexed family of sets as in $\mathbb{F}\textsc{am}$, and a morphism from $(J\,,\,Y)$ to $(I\,,\,X)$ consists of a function $e : J \to I$ 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. 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. We can then define the following two functors, forming an isomorphism of categories between $\mathbb{F}\textsc{ref}$ and $\mathbb{F}\textsc{am}$:

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

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

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

- Conversely, there is a functor $\text{FR}\textsc{ef}\text{C} : \mathsf{Functor}\ \mathbb{F}\textsc{am}\ \mathbb{F}\textsc{ref}$ whose object part is identity and whose morphism part is componentwise $\mathit{canonRef}$:

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

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

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

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

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

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

- Morphism composition is sequential composition $\_\odot\_$, which merges two successive batches of modifications in a straightforward way. There is also a family of **identity ornaments**, which simply use σ and v everywhere to express that a description is identical to itself, and can be proved to serve as identity of sequential composition. Their definitions are shown in Figure 4.4.

A functor Ind : $\mathsf{Functor}$ $\mathbb{O}$rn $\mathbb{F}$am can then be constructed, which gives the ordinary semantics of descriptions and ornaments: the object part of 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*.

$$
\begin{array}{l}
\mathbb{E}\text{-}\mathit{refl} : (\mathit{is} : \mathsf{List}\ I) \to \mathbb{E}\ \mathit{id}\ \mathit{is}\ \mathit{is} \\
\mathbb{E}\text{-}\mathit{refl}\ [\,] \qquad\quad = [\,] \\
\mathbb{E}\text{-}\mathit{refl}\ (i :: \mathit{is}) \quad = \mathsf{refl} :: \mathbb{E}\text{-}\mathit{refl}\ \mathit{is}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{idOrn} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) \to \mathsf{Orn}\ \mathit{id}\ D\ D \\
\mathit{idOrn}\ \{I\}\ D\ (\mathsf{ok}\ i) = \mathit{idROrn}\ (D\ i)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}\text{-}\mathit{trans} : \{I\ J\ K : \mathsf{Set}\}\ \{e : J \to I\}\ \{f : K \to J\} \to \\
\qquad\qquad \{\mathit{is} : \mathsf{List}\ I\}\ \{\mathit{js} : \mathsf{List}\ J\}\ \{\mathit{ks} : \mathsf{List}\ K\} \to \\
\qquad\qquad \mathbb{E}\ e\ \mathit{js}\ \mathit{is} \to \mathbb{E}\ f\ \mathit{ks}\ \mathit{js} \to \mathbb{E}\ (e \circ f)\ \mathit{ks}\ \mathit{is} \\
\mathbb{E}\text{-}\mathit{trans} \qquad\quad [\,] \qquad\qquad [\,] \qquad\qquad = [\,] \\
\mathbb{E}\text{-}\mathit{trans}\ \{e := e\}\ (\mathit{eeq} :: \mathit{eeqs})\ (\mathit{feq} :: \mathit{feqs}) = \mathit{trans}\ (\mathit{cong}\ e\ \mathit{feq})\ \mathit{eeq} :: \mathbb{E}\text{-}\mathit{trans}\ \mathit{eeqs}\ \mathit{feqs}
\end{array}
$$

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathit{scROrn} : \{I\ J\ K : \mathsf{Set}\}\ \{e : J \to I\}\ \{f : K \to J\} \to} \\
\multicolumn{6}{l}{\qquad\qquad \{D : \mathsf{RDesc}\ I\}\ \{E : \mathsf{RDesc}\ J\}\ \{F : \mathsf{RDesc}\ K\} \to} \\
\multicolumn{6}{l}{\qquad\qquad \mathsf{ROrn}\ e\ D\ E \to \mathsf{ROrn}\ f\ E\ F \to \mathsf{ROrn}\ (e \circ f)\ D\ F} \\
\mathit{scROrn}\ (\mathsf{v}\ \mathit{eeqs}) & (\mathsf{v}\ \mathit{feqs}) & = & \multicolumn{3}{l}{\mathsf{v}\ (\mathbb{E}\text{-}\mathit{trans}\ \mathit{eeqs}\ \mathit{feqs})} \\
\mathit{scROrn}\ (\mathsf{v}\ \mathit{eeqs}) & (\Delta\ T\ P) & = & \Delta[\,t : T\,] & \mathit{scROrn}\ (\mathsf{v}\ \mathit{eeqs}) & (P\ t) \\
\mathit{scROrn}\ (\sigma\ S\ O) & (\sigma\ .S\ P) & = & \sigma[\,s : S\,] & \mathit{scROrn}\ (O\ s) & (P\ s) \\
\mathit{scROrn}\ (\sigma\ S\ O) & (\Delta\ T\ P) & = & \Delta[\,t : T\,] & \mathit{scROrn}\ (\sigma\ S\ O) & (P\ t) \\
\mathit{scROrn}\ (\sigma\ S\ O) & (\nabla\ s\ P) & = & \nabla[\,s\,] & \mathit{scROrn}\ (O\ s) & P \\
\mathit{scROrn}\ (\Delta\ T\ O) & (\sigma\ .T\ P) & = & \Delta[\,t : T\,] & \mathit{scROrn}\ (O\ t) & (P\ t) \\
\mathit{scROrn}\ (\Delta\ T\ O) & (\Delta\ U\ P) & = & \Delta[\,u : U\,] & \mathit{scROrn}\ (\Delta\ T\ O) & (P\ u) \\
\mathit{scROrn}\ (\Delta\ T\ O) & (\nabla\ t\ P) & = & & \mathit{scROrn}\ (O\ t) & P \\
\mathit{scROrn}\ (\nabla\ s\ O) & P & = & \nabla[\,s\,] & \mathit{scROrn}\ O & P
\end{array}
$$

$$
\begin{array}{l}
\_\odot\_ : \{I\ J\ K : \mathsf{Set}\}\ \{e : J \to I\}\ \{f : K \to J\} \to \\
\qquad \{D : \mathsf{Desc}\ I\}\ \{E : \mathsf{Desc}\ J\}\ \{F : \mathsf{Desc}\ K\} \to \\
\qquad \mathsf{Orn}\ e\ D\ E \to \mathsf{Orn}\ f\ E\ F \to \mathsf{Orn}\ (e \circ f)\ D\ F \\
\_\odot\_\ \{f := f\}\ O\ P\ (\mathsf{ok}\ k) = \mathit{scROrn}\ (O\ (\mathsf{ok}\ (f\ k)))\ (P\ (\mathsf{ok}\ k))
\end{array}
$$

Figure 4.4 Definitions for identity ornaments and sequential composition of ornaments.

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

To translate $\mathbb{O}\textsc{rn}$ to $\mathbb{F}\textsc{ref}$, a naive way is to use the composite functor

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

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 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.

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

4.1.3 Isomorphisms

So far the categorical organisation offers no obvious benefits. It is for the ease of talking about mapping purely categorical structures between categories that we employ the categorical language. One simplest example of such purely categorical structures is **isomorphisms**: the type of isomorphisms between two objects X and Y in a category C is defined by

$$
\begin{array}{l}
\mathbf{record}\ \mathsf{Iso}\ C\ X\ Y \ : \ \mathsf{Set}\ \_\ \mathbf{where} \\
\quad \mathbf{field} \\
\qquad \mathit{to} \quad : \ X \Rightarrow Y
\end{array}
$$

$$\begin{aligned}
&\mathit{from} \;:\; Y \Rightarrow X\\
&\mathit{from\text{-}to\text{-}inverse} \;:\; \mathit{from} \cdot \mathit{to} \approx \mathit{id}\\
&\mathit{to\text{-}from\text{-}inverse} \;:\; \mathit{to} \cdot \mathit{from} \approx \mathit{id}
\end{aligned}$$

(We assume that, by introducing the category C in the text, there is implicitly a statement **open** Category C, so $\_{\Rightarrow}\_$ refers to Category.$\_{\Rightarrow}\_$ C and so on.) The relation $\_{\cong}\_$ we have been using is formally defined as Iso $\mathbb{F}$UN. Isomorphisms are preserved by functors, i.e.,

$$(F \;:\; \mathsf{Functor}\; C\; D) \to \mathsf{Iso}\; C\; X\; Y \to \mathsf{Iso}\; D\; (\mathit{object}\; F\; X)\; (\mathit{object}\; F\; Y)$$

which is proved by mapping all objects, morphisms, compositions, and equivalences in C appearing in the input isomorphism into D by F. (We **open** Functor so Functor.*object* can be simply referred to as *object* and so on.) This fact immediately tells us, for example, that when two ornaments are inverse to each other, so are their forgetful functions, by taking $F = \text{IND}$.

Another useful class of purely categorical structures are introduced next.

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 "strongly best" solution to some specification. More precisely: The specification is represented as a category whose objects are all possible solutions. A morphism from X to Y is evidence that Y is (non-strictly) "better" than X, and there can be more than one piece of such evidence. A "strongly best" solution is a **terminal object** in this category, meaning that it is "uniquely evidently better" than all objects in the category. Formally: an object Y 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 Y, i.e., the setoid *Morphism* X Y has a unique inhabitant:

$$\mathit{Terminal}\ C\ Y\ :\ \mathsf{Set}\ \_$$
$$\mathit{Terminal}\ C\ Y\ =\ (X\ :\ \mathit{Object}) \to \mathit{Singleton}\ (\mathit{Morphism}\ X\ Y)$$

where *Singleton* is defined by

$$\mathit{Singleton}\ :\ (S\ :\ \mathsf{Setoid}) \to \mathsf{Set}\ \_$$
$$\mathit{Singleton}\ S\ =\ \mathsf{Setoid}.\mathit{Carrier}\ S \times ((s\ t\ :\ \mathsf{Setoid}.\mathit{Carrier}\ S) \to s \approx_S t)$$

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:

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

Thus, to prove that two constructions are isomorphic, one way is 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 is to find suitable universal properties that characterise parallel composition, preparing for Sections 4.3.1 and 4.3.2.

Products

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

$$
\begin{array}{l}
\textbf{record}\ \mathsf{Span}\ C\ L\ R\ :\ \mathsf{Set}\ \_\ \textbf{where} \\
\quad \textbf{constructor}\ \mathsf{span} \\
\quad \textbf{field} \\
\qquad M\ :\ \mathit{Object} \\
\qquad l\ \ :\ M \Rightarrow L \\
\qquad r\ \ :\ M \Rightarrow R
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{record}\ \mathsf{SpanMorphism}\ C\ L\ R\ (s\ s'\ :\ \mathsf{Span}\ C\ L\ R)\ :\ \mathsf{Set}\ \_\ \textbf{where} \\
\quad \textbf{constructor}\ \mathsf{spanMorphism} \\
\quad \textbf{field} \\
\qquad m\ :\ \mathsf{Span}.M\ s \Rightarrow \mathsf{Span}.M\ s' \\
\qquad \mathit{triangle\text{-}l}\ :\ \mathsf{Span}.l\ s' \cdot m\ \approx\ \mathsf{Span}.l\ s \\
\qquad \mathit{triangle\text{-}r}\ :\ \mathsf{Span}.r\ s' \cdot m\ \approx\ \mathsf{Span}.r\ s
\end{array}
$$

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

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

where the two triangles are required to commute (i.e., *triangle-l* and *triangle-r* should hold). 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 now form a category of spans over L and R:

$$
\begin{array}{l}
\mathit{SpanCategory}\ C\ L\ R\ :\ \mathsf{Category} \\
\mathit{SpanCategory}\ C\ L\ R\ =\ \textbf{record}
\end{array}
$$

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

Note that the equivalence on span morphisms is defined to be the equivalence on the mediating morphism in C, ignoring the two triangular commutativity proofs. A product of L and R is then 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 in C that is more informative than both L and R.

Slice categories

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 $\mathbb{O}$RN. We are thus directed to construct a category whose objects are morphisms in an ambient category C, so when we use $\mathbb{O}$RN 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 in the ambient category 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
        { Carrier = SliceMorphism C B s s′
        ; _≈_     = λ f g ↦ SliceMorphism.m f ≈ SliceMorphism.m g
        ; proofs of laws }
  ; proofs of laws }
```

Objects in a slice category are thus morphisms with a common target, and when the ambient category is Orn, they are exactly the compatible ornaments that can be composed in parallel.

Pullbacks

We have arrived at the characterisation of parallel composition as a product in a slice category on top of ORN. 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 : 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

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

In a square q, we will refer to the object Slice.T (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

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

then a pullback of f and g is a terminal object in the square category over f and g — indeed, *Product* (*SliceCategory* C X) f g is definitionally equal to *Terminal* (*SquareCategory* 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:

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

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

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

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

Like isomorphisms, we can talk about preservation of pullbacks: any functor $F : \mathsf{Functor}\ C\ D$ maps a square in C into one in D; if the resulting square in D is a pullback whenever the input square in C is, then F is said to be **pullback-preserving**. Formally:

$$\begin{array}{l} \mathit{Pullback\text{-}Preserving}\ F\ : \\ \quad \{B\ :\ \mathsf{Category}.\mathit{Object}\ C\}\ \{f\ g\ :\ \mathsf{Slice}\ C\ B\}\ (s\ :\ \mathsf{Square}\ C\ f\ g) \rightarrow \\ \quad \mathit{Pullback}\ C\ f\ g\ s \rightarrow \mathit{Pullback}\ D\ (\mathit{object}\ (\mathit{SliceMap}\ F)\ f)\ (\mathit{object}\ (\mathit{SliceMap}\ F)\ g) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{object}\ (\mathit{SquareMap}\ F)\ s) \end{array}$$

where

$$\begin{array}{lll} \mathit{SliceMap} & : & (F\ :\ \mathsf{Functor}\ C\ D) \rightarrow \\ & & \mathsf{Functor}\ (\mathit{SliceCategory}\ C\ B)\ (\mathit{SliceCategory}\ D\ (\mathit{object}\ F\ B)) \\ \mathit{SquareMap} & : & (F\ :\ \mathsf{Functor}\ C\ D) \rightarrow \\ & & \mathsf{Functor}\ (\mathit{SquareCategory}\ C\ f\ g) \\ & & \qquad (\mathit{SquareCategory}\ D\ (\mathit{object}\ (\mathit{SliceMap}\ F)\ f) \\ & & \qquad\qquad\qquad\qquad (\mathit{object}\ (\mathit{SliceMap}\ F)\ g)) \end{array}$$

are straightforward liftings of functors on ambient categories to functors on slice and square categories. Unlike isomorphisms, pullbacks are not preserved by all functors, but the functors $\mathrm{I}\textsc{nd} : \mathsf{Functor}\ \mathbb{O}\textsc{rn}\ \mathbb{F}\textsc{am}$ and $\mathrm{C}\textsc{om} : \mathsf{Functor}\ \mathbb{F}\textsc{am}\ \mathbb{F}\textsc{un}$ are pullback-preserving, which we use below.

Parallel composition as a pullback

For any $O : \mathsf{Orn}\ e\ D\ E$ 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 $\mathbb{O}\textsc{rn}$ is a pullback:

$$\begin{tikzcd}[column sep=huge, row sep=huge]
e \bowtie f \mathbin{,} \lfloor O \otimes P \rfloor \arrow[r, "\mathit{outr}_{\bowtie} \mathbin{,} \mathit{diffOrn\text{-}r}\ O\ P"] \arrow[d, "\mathit{outl}_{\bowtie} \mathbin{,} \mathit{diffOrn\text{-}l}\ O\ P"'] \arrow[dr, "\mathit{pull}, \lceil O \otimes P \rceil" description] \arrow[dr, phantom, very near start, "\lrcorner"] & K \mathbin{,} F \arrow[d, "f \mathbin{,} P"] \\
J \mathbin{,} E \arrow[r, "e \mathbin{,} O"'] & I \mathbin{,} D
\end{tikzcd} \tag{4.2}$$

We assert that the square is a pullback by marking its vertex with "$\lrcorner$". The AGDA term for the square is

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

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 $\mathit{OrnEq}\ (P \odot \mathit{diffOrn\text{-}r}\ O\ P)\ \lceil O \otimes P \rceil$, both of which can be discharged. Comparing the commutative diagram (4.2) and the AGDA term *pc-square O P*, 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 O P* 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. A proof sketch of (4.2) is deferred to Chapter 6. The pullback property (4.2) by itself is not too useful in this chapter, though: $\mathbb{O}\textsc{rn}$ is a quite restricted category, so a universal property established in $\mathbb{O}\textsc{rn}$ has limited applicability. Instead, we are more interested in the following two pullback properties: the image of (4.2) under Ind in $\mathbb{F}$am:

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

and the image of (4.3) under Com in $\mathbb{F}$un:

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

Both (4.3) and (4.4) are indeed pullbacks by pullback preservation of Ind and Com.

4.3 Consequences

Characterising parallel composition as a pullback immediately allows us to instantiate standard categorical results, like commutativity $(\_ \, , \lfloor O \otimes P \rfloor) \cong (\_ \, , \lfloor P \otimes O \rfloor)$ and associativity $(\_ \, , \lfloor \lceil O \otimes P \rceil \otimes Q \rfloor) \cong (\_ \, , \lfloor O \otimes \lceil P \otimes Q \rceil \rfloor)$ up to isomorphism in $\mathbb{O}$rn (which can then be transferred by isomorphism preservation to $\mathbb{F}$am, for example).

4.3.1 The ornamental conversion isomorphisms

We restate the ornamental conversion isomorphisms as follows: for any ornament O : Orn $e \; D \; E$ where D : Desc I and E : 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 (4.4) in $\mathbb{F}$UN can help.

- Set-theoretically, the vertex of a pullback of two functions $f : A \to C$ and $g : B \to C$ is isomorphic to $\Sigma[p : A \times B]\ f\ (\mathit{outl}\ p) \equiv g\ (\mathit{outr}\ p)$; the more information f and g carries into C, the stronger the equality constraint. An extreme case is when C is just B and g is identity (which retains all information): the equality constraint reduces to $f\ (\mathit{outl}\ p) \equiv \mathit{outr}\ p$, so the second component of p is completely determined by the first component, and thus the vertex is isomorphic to just A. The same situation happens for the following pullback square:

$$\begin{tikzcd}[column sep=huge, row sep=large]
\Sigma\ J\ (\mu\ E) \arrow[r, "(e * \mathit{forget}\ O) \vartriangle (\mathit{singleton} \circ \mathit{forget}\ O \circ \mathit{outr})"] \arrow[d, "\mathit{id}"'] \arrow[dr, phantom, very near start, "\lrcorner"] \arrow[dr, "e * \mathit{forget}\ O" description] & \Sigma\ (\Sigma\ I\ (\mu\ D))\ (\mu\ \lfloor S \rfloor) \arrow[d, "\mathit{outl} * \mathit{forget}\ \lceil S \rceil"] \\
\Sigma\ J\ (\mu\ E) \arrow[r, "e * \mathit{forget}\ O"'] & \Sigma\ I\ (\mu\ D)
\end{tikzcd} \tag{4.6}$$

  Since singleton ornamentation does not add information to a datatype, the vertical slice on the right-hand side

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

  retains all information like an identity function, and thus behaves like a "multiplicative unit" (viewing pullbacks as products of slices): any (compatible) slice s' alone gives rise to a product of s and s'. In particular, we can use 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

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

- 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:

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

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

- The above 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

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

  for all $j\ :\ J$. Finally, observe that a canonical element of type $\mathit{outl}_{\bowtie}^{\ -1}\ j$ 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).

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

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

both indexed by $\mathit{pull} \bowtie \mathit{outl}$ 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 $(\mathit{pull} \bowtie \mathit{outl}\,,\, M)$ and $(\mathit{pull} \bowtie \mathit{outl}\,,\, N)$ whose index part (i.e., the isomorphism obtained under the functor FAMF) is identity. Thus we seek to prove that both $(\mathit{pull} \bowtie \mathit{outl}\,,\, M)$ and $(\mathit{pull} \bowtie \mathit{outl}\,,\, N)$ are vertices of pullbacks of the same slices.

- We look at $(\mathit{pull} \bowtie \mathit{outl}\,,\, N)$ 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 <u>singleton</u> set $\mu \lfloor S \rfloor \ (i\ , x)$, as shown in the following diagram:

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

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:

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

- Next we prove that $(pull \bowtie outl\ , M)$ 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 $Y \boxtimes Z \Rightarrow Z$.

$$\begin{tikzcd}
X & & X \boxtimes Y \arrow[ll] \arrow[rr] & & Y \\
& & (X \boxtimes Y) \boxtimes Z \arrow[u] \arrow[dl, dashed] \arrow[dr, dashed] \arrow[dd] & & \\
& X \boxtimes Z \arrow[ul] \arrow[dr] & & Y \boxtimes Z \arrow[ur] \arrow[dl] & \\
& & Z & &
\end{tikzcd}$$

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:

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

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 $(\mathit{pull} \bowtie \mathit{outl}\ , M)$.

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

4.4 Discussion

The categorical organisation of the ornament–refinement framework effectively summarises various constructions in the framework under the succinct categorical language. For example, the functor Ind from $\mathbb{O}$rn to $\mathbb{F}$am is itself a summary of the following:

- the least fixed-point operation on descriptions (the object part of the functor),
- the forgetful functions derived from ornaments (the morphism part of the functor),
- the equivalence on ornaments, which implies extensional equality on ornamental forgetful functions (since functors respect equivalence),
- the identity ornaments, whose forgetful functions are extensionally equal to identity functions (since functors preserve identity),
- sequential composition of ornaments, and the fact that the forgetful function for any sequentially composed ornament $O \odot P$ is extensionally equal to the composition of the forgetful functions for O and P (since functors preserve composition).

Most importantly, a categorical pullback structure emerges from the framework, which gives a macroscopic meaning to the microscopic type-theoretical definition of parallel composition (thus ensuring that the definition is not an arbitrary one), and enables constructions of the ornamental conversion isomorphisms and the modularity isomorphisms on a more abstract level. Compared to the constructions of the isomorphisms of Ko and Gibbons [2013] using datatype-generic induction, the constructions presented in this chapter offer more insights and are easier to understand: after establishing the pullback properties of parallel composition, at the root of the ornamental conversion isomorphisms is the intuition that singleton ornamentation does not add information, and the modularity isomorphisms stem from the fact that the pointwise conjunction of optimised predicates trivially extends to a pullback. Also, the categorical constructions are impervious to change of representation of the universes of descriptions and ornaments; modification to the universes only affects constructions logically prior to the pullback property (4.3). This statement is empirically verified — the representation of the universes really had to be changed once after carrying out the categorical constructions.

Dagand and McBride [2013] provided a purely categorical treatment of ornaments using fibred category theory, which is quite independent of the development in this chapter, though. They established correspondences between descriptions and polynomial functors [Gambino and Kock, 2010] and between ornaments and cartesian morphisms, and sketched how several operations on ornaments correspond to certain categorical notions (including pullbacks), all of which are ultimately based on the abstract notion of locally cartesian closed categories. This chapter, on the other hand, merely uses a lightweight categorical language for organising various type-theoretic constructions, and the reasoning is still essentially type-theoretical, rather than insisting on purely categorical arguments. Methodologically, there is another notable difference: Dagand and McBride distinguish "software" (e.g., descriptions and ornaments) and "mathematics" (e.g., polynomial functors and cartesian morphisms) and then make a connection between them, so they can use mathematical notions as inspiration for software constructs. In the light of the propositions-as-types principle, though, a further step should be taken: rather than merely making a connection between software artifacts that have the necessary level of detail and mathematical objects that possess desired abstract properties, we should design the software artifacts such that they satisfy the abstract properties themselves, all expressed in one uniform language, so the detail of the artifacts can be effectively managed by reasoning in terms of the abstract properties. In this spirit, category theory is regarded in this chapter as providing a set of abstractions that are internalised in type theory, as opposed to being an independent formalism. Ornaments are complex, but the complexity is necessary (as justified in Section 3.5) and hence can only be somehow managed rather than being eliminated. Fortunately, ornaments exhibit a useful categorical structure, and we are able to tame the complexity of ornaments by reasoning abstractly in terms of this categorical structure.

The results of this chapter are completely formalised in AGDA. The setoid approach to managing morphism equivalence works reasonably well for this chapter, being able to express extensional equality (e.g., in $\mathbb{F}$UN and $\mathbb{F}$AM), proof irrelevance (e.g., in *SpanCategory* and *SliceCategory*), and layered definitions of equality (e.g., morphism equivalence in *SquareCategory* is ultimately defined in terms of morphism equivalence in the ambient category). We will see, however, that the approach works only because the formalised theory is still simple enough — Chapter 6 deals with more complicated categorical notions, and the limitation of the approach will become clearer then. Even in this chapter, formalisation in general is already difficult due to AGDA's intensionality, and the reasoning presented cannot always be faithfully encoded: For the last two steps of the argument in Section 4.3.1, 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 containing plenty of suspended *subst*s. If these isomorphisms are used to construct the refinement family in the morphism part of RSEM, it would be terribly difficult to prove that the morphism part of RSEM preserves equivalence. The actual formalisation circumvents the problem by fusing the lemma and the change of variables into one step to get a clean AGDA definition such that RSEM preserves equivalence automatically. While this kind of change of arguments for formalisation seems to be the norm (see, e.g., Avigad et al. [2007, Section 4.5]), it can occur more frequently in dependently typed programming due to the requirement that the constructed terms have to be "pretty" if they are to be referred to in later types, so even a powerful tactic language for constructing proofs probably would not help much, since we cannot control the form of the resulting proofs. More experience in tackling this kind of proof-relevant formalisation is needed.

Chapter 5

Relational algebraic ornamentation

This chapter turns to the **synthetic** direction of the interconnection between internalism and externalism. As stated in Section 2.5, internalist types can be hard to read and write, and it would be helpful to be able to switch to an alternative language for understanding and deriving internalist types. The alternative language adopted in this chapter is the **relational** language (Section 5.1), of which Bird and de Moor [1997] gave an authoritative account. Unlike the datatype declaration language, using relations we can give concise yet computationally intuitive specifications, which are amenable to manipulation by algebraic laws and theorems. A particularly expressive relational construction is the **relational fold**, and when fixing a basic datatype and casting the relational fold as the externalist predicate, we can synthesise a corresponding internalist datatype on the other side of the conversion isomorphism. More specifically, every relational algebra gives rise to an **algebraic ornamentation** (Section 5.2), whose optimised predicate (Section 3.3.1) can be swapped (Section 3.3.2) for the relational fold with the algebra. Specifications involving relational folds can then be met by constructing internalist programs whose types involve corresponding algebraic ornamented datatypes. Several examples are given in Section 5.3, followed by some discussion in Section 5.4.

5.1 Relational programming in Agda

Functional programs are known for their amenability to algebraic calculation (see, e.g., Backus [1978] and Bird [2010]), leading to one form of program correctness by construction: 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** (see, e.g., Bird [1996]). 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 inhabitant of A to a subset of B. We define subsets by

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

That is, a subset $s : \mathcal{P}\, A$ is a characteristic function that assigns a type to each inhabitant 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 inhabitant 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 inhabitant of A. Since $\top$ is inhabited, *any* can produce any inhabitant 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

  $$\begin{aligned}&\_{\gg\!=}\_ \;:\; \{A\; B \;:\; \mathsf{Set}\} \to \mathcal{P}\,A \to (A \to \mathcal{P}\,B) \to \mathcal{P}\,B\\&\_{\gg\!=}\_\; \{A\}\; s\, f \;=\; \lambda\, b \mapsto \Sigma[\, a : A\,]\;\; s\; a \times f\; a\; b\end{aligned}$$

  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 inhabitants of A that belong to s; that is, an inhabitant $b \;:\; B$ is a member of $s \gg\!= f$ exactly when there exists some $a \;:\; A$ belonging to s such that b is a member of $f\; a$.

(We omit the proofs that the two combinators satisfy the (relative) monad laws up to pointwise isomorphism.) On top of *return* and $\_{\gg\!=}\_$, the functorial map on $\mathcal{P}$ 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 $\mathscr{P}$ as applications of pure functions to effectful arguments.)

We define a relation between two families of sets as a family of relations between corresponding sets in the families:

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

which is the usual generalisation of $\_{\rightrightarrows}\_$ to allow nondeterminacy. Below we define several relational operators that we will need.

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

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

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

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

- 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.

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

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

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

- 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:

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

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

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

Figure 5.1 Definition for relators.

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:

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

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,

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

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

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

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

$$\_{\simeq}\_ \;:\; \{I \;:\; \mathsf{Set}\}\;\{X\;Y \;:\; I \rightarrow \mathsf{Set}\}\;(R\;S \;:\; X \rightsquigarrow Y) \rightarrow \mathsf{Set}$$
$$R \simeq S \;=\; (R \subseteq S) \times (R \supseteq S)$$

where $R \supseteq S$ is defined to be $S \subseteq R$ as usual. For example, a relator preserves identity and composition, i.e.,

$$\mathbb{R}\;D\;\mathit{idR} \simeq \mathit{idR} \qquad \text{and} \qquad \mathbb{R}\;D\;(R \bullet S) \simeq \mathbb{R}\;D\;R \bullet \mathbb{R}\;D\;S$$

and is monotonic, i.e.,

$$\mathbb{R}\;D\;R \subseteq \mathbb{R}\;D\;S \qquad \text{whenever} \quad R \subseteq S$$

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

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

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. (For example, let

$$D \;=\; \lambda\;\{\;\blacksquare\;\mapsto\;\mathsf{v}\;(\blacksquare :: \blacksquare :: [\,])\;\}\;:\;\mathsf{Desc}\;\top$$

be a trivially indexed description with two recursive positions, and define

$$\mathit{plus} \;=\; \mathit{fun}\;(\lambda\;\{\;(x\,,y\,,\blacksquare)\;\mapsto\;x + y\;\})\;:\;\mathbb{F}\;D\;\mathsf{Nat} \rightsquigarrow \mathsf{Nat}$$

Then *plus* is monotonic on

$$\mathit{leq} \;=\; \lambda\,x\,y \;\mapsto\; y \leqslant x \;:\; \mathit{const}\;\mathsf{Nat} \rightsquigarrow \mathit{const}\;\mathsf{Nat}$$

which maps a natural number x to any natural number y that is at most x. Pointwise, the monotonicity statement expands to

$$(x\;y\;x'\;y' \;:\; \mathsf{Nat}) \rightarrow (x \leqslant x') \times (y \leqslant y') \rightarrow x + y \leqslant x' + y'$$

i.e., addition is monotonic on its two arguments.) In the context of optimisation problems, monotonicity can be used to capture the **principle of optimality**, as

mutual

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

Figure 5.2 Definition of relational folds.

will be shown in Section 5.3.3. Section 5.3.1 contains some simple relational calculations involving the above properties.

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 the "banana bracket" [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 (cf. the definition of *fold* in Figure 2.1). For example, the relational fold on lists is, up to isomorphism,

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

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

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

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

5.2 Definition of algebraic ornamentation

We now turn to algebraic ornamentation, the key construct that bridges internalist and relational programming, and look at a special case first. Let

$$R \,:\, \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ X) \rightsquigarrow \mathit{const}\ X \qquad \text{where} \quad X \,:\, \mathsf{Set}$$

be a relational algebra for lists. We can define a datatype of "algebraically ornamented lists" as

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

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

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

A simple argument by induction shows that $\mathsf{AlgListP}\ A\ R\ x\ \mathit{as}$ is in fact isomorphic to $(\![\, R \,]\!)\ \mathit{as}\ x$ for any $\mathit{as} \,:\, \mathsf{List}\ A$ and $x \,:\, X$ — that is, we can do predicate swapping for the refinement semantics of the ornament from lists to algebraically ornamented lists (Section 3.3). Thus we get the conversion isomorphisms

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

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: Let *length-alg* be the ornamental algebra derived from the ornament from natural numbers to lists (so $\mathit{length} = \mathit{fold}\ \mathit{length\text{-}alg}$).

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

$$
\begin{aligned}
&\mathit{algOD} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{J : I \to \mathsf{Set}\} \to \\
&\qquad (\mathbb{F}\ D\ J \rightsquigarrow J) \to \mathsf{OrnDesc}\ (\Sigma\ I\ J)\ \mathit{outl}\ D \\
&\mathit{algOD}\ D\ R\ (\mathsf{ok}\ (i\ ,j)) \ = \ \mathit{algROD}\ (D\ i)\ (\lambda\ \mathit{js} \mapsto R\ \mathit{js}\ j)
\end{aligned}
$$

Figure 5.3 Definitions for algebraic ornamentation.

$$
\begin{aligned}
&\mathit{length\text{-}alg} : \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat} \\
&\mathit{length\text{-}alg} \ = \ \mathit{ornAlg}\ \lceil \mathit{NatD\text{-}ListD}\ A \rceil
\end{aligned}
$$

Then $\mathsf{AlgList}\ A\ (\mathit{fun}\ \mathit{length\text{-}alg})$ is exactly $\mathsf{Vec}'\ A$ in Section 2.4.1. By (5.1) we have

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

from which we can further derive

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

by *fun-preserves-fold*.

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

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

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

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

which is proved by induction on d. These isomorphisms give rise to a swap family

$$\mathit{algOD\text{-}FSwap}\ D\ R : \mathsf{FSwap}\ (\mathit{RSem}\ \lceil \mathit{algOD}\ D\ R \rceil)$$

such that $\mathsf{Swap}.P\ (\mathit{algOD\text{-}FSwap}\ D\ R\ (\mathsf{ok}\ (i\ ,\ x)))\ =\ \lambda\, d\ \mapsto\ (\!\lbrack R \rbrack\!)\ d\ x$, so we arrive at the following conversion isomorphisms

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

The definition of *algOD*, shown in Figure 5.3, is an adaptation and generalisation of McBride's original definition of functional algebraic ornamentation [2011]. Roughly speaking, it retains (in the σ case of *algROD*) all the fields of the base description and inserts (in the v case of *algROD*) 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).

Algebraic ornamentation is a convenient method for adding new indices to inductive families. (We will see in Chapter 6 that it is actually a canonical way to refine inductive families by ornamentation.) Most importantly, the conversion isomorphisms (5.2) state clearly what the new indices mean in terms of relations. We can thus easily translate relational expressions into internalist types for type-directed programming, as demonstrated in the next section.

5.3 Examples

We give three examples about three relational theorems.

- Section 5.3.1 shows how the **Fold Fusion Theorem** [Bird and de Moor, 1997, Section 6.2] gives rise to conversion functions between algebraically ornamented datatypes.
- Section 5.3.2 implements the **Streaming Theorem** [Bird and Gibbons, 2003, Theorem 30] as an internalist program, whose type directly corresponds to

the "metamorphic" specification stated by the theorem.

- Section 5.3.3 uses the **Greedy Theorem** [Bird and de Moor, 1997, Theorem 10.1] to nontrivially derive a suitable type for an internalist program that solves the **minimum coin change problem**.

5.3.1 The Fold Fusion Theorem

The statement of the **Fold Fusion Theorem** [Bird and de Moor, 1997, Section 6.2] is as follows: Let $D : \mathsf{Desc}\ I$ 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.,

$$R \bullet S \simeq T \bullet \mathbb{R}\, D\, R$$

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

$$R \bullet (\!\![\, S \,]\!\!) \simeq (\!\![\, T \,]\!\!)$$

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 variation is

$$R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R \quad \rightarrow \quad R \bullet (\!\![\, S \,]\!\!) \subseteq (\!\![\, T \,]\!\!) \tag{5.3}$$

and the other variation simply reverses the direction of inclusion:

$$R \bullet S \supseteq T \bullet \mathbb{R}\, D\, R \quad \rightarrow \quad R \bullet (\!\![\, S \,]\!\!) \supseteq (\!\![\, T \,]\!\!) \tag{5.4}$$

Both of them roughly state that a fold can be conditionally transformed into another. Since algebraically ornamented datatypes are datatypes with constraints expressed as a fold, we should be able to transform these constraints by the Fold Fusion Theorem while leaving the underlying data unchanged, thus converting one algebraically ornamented datatype into another.

We look at (5.3) first. Assume that we have a proof of the antecedent

$$\mathit{fcond}_{\subseteq} : R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R$$

Expanding the conclusion of (5.3) pointwise: if $d : \mu\ D\ i$ folds to $x : X\ i$ by S, which is "relaxed" to $y : Y\ i$ by R, then d folds to y by T. This can be translated

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

$$
\mathbf{syntax}\;\mathit{new}\text{-}\Sigma\;I\;(\lambda\;i \to u) \;=\; \Sigma^{+}[\,i : I\,]\;u
$$

$$
\begin{aligned}
&\_\times^{+}\_ \;:\; \{X\;Y \;:\; \mathsf{Set}\} \to \mathsf{Upgrade}\;X\;Y \to (Z \;:\; \mathsf{Set}) \to \mathsf{Upgrade}\;X\;(Y \times Z) \\
&u \times^{+} Z \;=\; \mathbf{record}\;\{\;P \;=\; \lambda\,x \mapsto \mathsf{Upgrade}.P\;u\;x \times Z \\
&\qquad\qquad\qquad ;\;C \;=\; \lambda\;\{\;x\;(y\,,z) \;\mapsto\; \mathsf{Upgrade}.C\;u\;x\;y\;\} \\
&\qquad\qquad\qquad ;\;u \;=\; \lambda\;\{\;x\;(p\,,z) \;\mapsto\; \mathsf{Upgrade}.u\;u\;x\;p\;,z\;\} \\
&\qquad\qquad\qquad ;\;c \;=\; \lambda\;\{\;x\;(p\,,z) \;\mapsto\; \mathsf{Upgrade}.c\;u\;x\;p\;\}\;\}
\end{aligned}
$$

Figure 5.4 Two additional upgrade combinators.

to a conversion function from a datatype algebraically ornamented with S to one with T:

$$
\begin{aligned}
&\mathit{fusion\text{-}conversion}_{\subseteq}\;D\;R\;S\;T\;\mathit{fcond}_{\subseteq}\;: \\
&\quad \{i \;:\; I\}\;(x \;:\; X\;i) \to \mu\;\lfloor \mathit{algOD}\;D\;S \rfloor\;(i\,,x) \to \\
&\quad (y \;:\; Y\;i) \to R\;x\;y \to \mu\;\lfloor \mathit{algOD}\;D\;T \rfloor\;(i\,,y)
\end{aligned}
$$

This function does not alter the underlying $(\mu\;D)$–data, which can be easily expressed by upgrading (Section 3.1.2) the identity function on $\mu\;D$ to its type. We thus write the following upgrade

$$
\begin{aligned}
&\mathit{upg}_{\subseteq}\;:\;\mathsf{Upgrade}\;(\{i \;:\; I\} \to \mu\;D\;i \to \mu\;D\;i) \\
&\qquad\qquad\qquad (\{i \;:\; I\}\;\{x \;:\; X\;i\} \to \mu\;\lfloor \mathit{algOD}\;D\;S \rfloor\;(i\,,x) \to \\
&\qquad\qquad\qquad\; \{y \;:\; Y\;i\} \to R\;x\;y \to \mu\;\lfloor \mathit{algOD}\;D\;T \rfloor\;(i\,,y)) \\
&\mathit{upg}_{\subseteq} \;=\; \forall[[\,i : I\,]]\;\forall^{+}[[\,x : X\;i\,]]\;\mathit{rS}\;i\;x \rightharpoonup \\
&\qquad\quad \forall^{+}[[\,y : Y\;i\,]]\;\forall^{+}[\,\_ : R\;x\;y\,]\;\mathit{toUpgrade}\;(\mathit{rT}\;i\;y) \\
&\quad\mathbf{where} \\
&\qquad \mathit{rS} \;:\; (i \;:\; I)\;(x \;:\; X\;i) \to \mathsf{Refinement}\;(\mu\;D\;i)\;(\mu\;\lfloor \mathit{algOD}\;D\;S \rfloor\;(i\,,x)) \\
&\qquad \mathit{rS}\;i\;x \;=\; \mathit{toRefinement}\;(\mathit{algOD\text{-}FSwap}\;D\;S\;(\mathsf{ok}\;(i\,,x)))
\end{aligned}
$$

$$rT : (i : I)\ (y : Y\ i) \to \mathsf{Refinement}\ (\mu\ D\ i)\ (\mu\ \lfloor algOD\ D\ T \rfloor\ (i\ , y))$$
$$rT\ i\ y\ =\ toRefinement\ (algOD\text{-}FSwap\ D\ T\ (\mathsf{ok}\ (i\ , y)))$$

and implement the conversion function by

$$\mathsf{Upgrade}.u\ upg_{\subseteq}\ id\ \{\ \}_0$$

Goal 0 demands a promotion proof of type

$$\{i : I\}\ \{x : X\ i\}\ (d : \mu\ D\ i) \to (\!|\ S\ |\!)\ d\ x \to \{y : Y\ i\} \to R\ x\ y \to (\!|\ T\ |\!)\ (id\ d)\ y$$

which is exactly the pointwise expansion of the conclusion of (5.3). The coherence property

$$\begin{aligned}
&\{i : I\}\ \{x : X\ i\}\ (d : \mu\ D\ i)\ (d' : \mu\ \lfloor algOD\ D\ S \rfloor\ (i\ , x)) \to \\
&\quad forget\ \lceil algOD\ D\ S \rceil\ d'\ \equiv\ d \to \\
&\{y : Y\ i\} \to R\ x\ y \to \\
&\quad forget\ \lceil algOD\ D\ T \rceil\ (fusion\text{-}conversion_{\subseteq}\ D\ R\ S\ T\ fcond_{\subseteq}\ d')\ \equiv\ id\ d
\end{aligned}$$

then states that the conversion function transforms the underlying $(\mu\ D)$-data in the same way as *id*, i.e., it leaves the underlying data unchanged. Similarly for (5.4), assuming that we have

$$fcond_{\supseteq} : R \bullet S\ \supseteq\ T \bullet \mathbb{R}\ D\ R$$

we should be able to construct the conversion function

$$\begin{aligned}
&fusion\text{-}conversion_{\supseteq}\ D\ R\ S\ T\ fcond_{\supseteq} : \\
&\quad \{i : I\}\ (y : Y\ i) \to \mu\ \lfloor algOD\ D\ T \rfloor\ (i\ , y) \to \\
&\quad \Sigma[\, x : X\ i \,]\ \mu\ \lfloor algOD\ D\ S \rfloor\ (i\ , x) \times R\ x\ y
\end{aligned}$$

as an upgraded version of the identity function with the upgrade

$$\begin{aligned}
&upg_{\supseteq} : \mathsf{Upgrade}\ (\{i : I\} \to \mu\ D\ i \to \mu\ D\ i) \\
&\qquad\qquad (\{i : I\}\ \{y : Y\ i\} \to \mu\ \lfloor algOrn\ D\ T \rfloor\ (i\ , y) \to \\
&\qquad\qquad\ \ \Sigma[\, x : X\ i \,]\ \mu\ \lfloor algOrn\ D\ S \rfloor\ (i\ , x) \times R\ x\ y) \\
&upg_{\supseteq}\ =\ \forall^{+}[[\, i : I \,]]\ \forall^{+}[[\, y : Y\ i \,]]\ rT\ i\ y \rightharpoonup \\
&\qquad\qquad \Sigma^{+}[\, x : X\ i \,]\ toUpgrade\ (rS\ i\ x) \times^{+} R\ x\ y \\
&\quad \mathbf{where} \\
&\qquad rS : (i : I)\ (x : X\ i) \to \mathsf{Refinement}\ (\mu\ D\ i)\ (\mu\ \lfloor algOrn\ D\ S \rfloor\ (i\ , x)) \\
&\qquad rS\ i\ x\ =\ toRefinement\ (algOD\text{-}FSwap\ D\ S\ (\mathsf{ok}\ (i\ , x)))
\end{aligned}$$

$$rT \;:\; (i \;:\; I)\;(y \;:\; Y\;i) \rightarrow \mathsf{Refinement}\;(\mu\;D\;i)\;(\mu\;\lfloor algOrn\;D\;T \rfloor\;(i\,,y))$$
$$rT\;i\;y \;=\; toRefinement\;(algOD\text{-}FSwap\;D\;T\;(\mathsf{ok}\;(i\,,y)))$$

in which we need two new combinators to deal with upgrading to product types, which are defined in Figure 5.4.

For a simple application, 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

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

where $geq \;=\; leq^{\circ} \;:\; const\;\mathsf{Nat} \rightsquigarrow const\;\mathsf{Nat}$ maps a natural number x to any natural number that is at least x. The conversion isomorphisms (5.2) specialise for BVec to

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

for all $m \;:\; \mathsf{Nat}$. But is BVec really the bounded vectors? Indeed it is, because we can deduce

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

by Fold Fusion. The fusion condition is

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

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

$$
\begin{array}{ll}
 & geq \bullet fun\;length\text{-}alg \\
\subseteq & \quad \{\,\text{identity}\,\} \\
 & geq \bullet fun\;length\text{-}alg \bullet idR \\
\subseteq & \quad \{\,\text{relator preserves identity}\,\} \\
 & geq \bullet fun\;length\text{-}alg \bullet \mathbb{R}\;(ListD\;A)\;idR \\
\subseteq & \quad \{\,geq\;\text{reflexive; relator is monotonic}\,\} \\
 & geq \bullet fun\;length\text{-}alg \bullet \mathbb{R}\;(ListD\;A)\;geq
\end{array}
$$

And from right to left:

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

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 $\mathit{fusion\text{-}conversion}_{\subseteq}$ and $\mathit{fusion\text{-}conversion}_{\supseteq}$. Hence we get two conversions, one of type

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

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

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

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.

5.3.2 The Streaming Theorem for list metamorphisms

A **metamorphism** [Gibbons, 2007b] 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 algebraically 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

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

  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:

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

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

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

$$fun\ (foldl\ f\ x)\ \simeq\ fun\ (\lambda\, h\ \mapsto\ h\ x)\ \bullet\ (\!|\, fun\ (foldl\text{-}alg\ f)\,|\!) \tag{5.5}$$

- The unfold part is approximated by the converse of a relational fold: given a list coalgebra $g\ :\ const\ X \rightrightarrows \mathbb{F}\ (ListD\ B)\ (const\ X)$ 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.

  $$(\!|\, fun\ g^{\circ}\,|\!)^{\circ}\ :\ const\ X \rightsquigarrow const\ (\mathsf{List}\ A)$$

  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 $\mathsf{List}\ B$

$$g\ :\ const\ X \rightrightarrows \mathbb{F}\ (ListD\ B)\ (const\ X)$$

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:

$$meta\ f\ g\ x\ =\ (\!|\, fun\ g^{\circ}\,|\!)^{\circ}\ \bullet\ fun\ (foldl\ f\ x)\ :\ const\ (\mathsf{List}\ A) \rightsquigarrow const\ (\mathsf{List}\ B)$$

Next we devise a type for the streaming algorithm that fully guarantees its correctness. By (5.5), the specification $meta\ f\ g\ x$ is equivalent to

$$(\!|\, fun\ g^{\circ}\,|\!)^{\circ}\ \bullet\ fun\ (\lambda\, h\ \mapsto\ h\ x)\ \bullet\ (\!|\, fun\ (foldl\text{-}alg\ f)\,|\!)$$

Inspecting the above relation, we see that a conforming program takes a $\mathsf{List}\ A$ that folds to some $h\ :\ X \to X$ with $fun\ (foldl\text{-}alg\ f)$ and computes a $\mathsf{List}\ B$ that folds to $h\ x\ :\ X$ with $fun\ g^{\circ}$. We thus implement the streaming algorithm by

$$\begin{aligned} stream\ f\ g\ :\ & (x\ :\ X)\ \{h\ :\ X \to X\} \to \\ & \mathsf{AlgList}\ A\ (fun\ (foldl\text{-}alg\ f))\ h \to \mathsf{AlgList}\ B\ (fun\ g^{\circ})\ (h\ x) \end{aligned}$$

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 $\mathsf{AlgList}$ datatypes involved:

$$\begin{array}{l} \mathit{consumption\text{-}iso}\ : \\ \quad (h\ :\ X \to X) \to \\ \quad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h\ \cong\ \Sigma[\ \mathit{as} : \mathsf{List}\ A\ ]\ \mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as}\ \equiv\ h \\ \mathit{production\text{-}iso}\ : \\ \quad (x\ :\ X) \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ x\ \cong\ \Sigma[\ \mathit{bs} : \mathsf{List}\ B\ ]\ (\![\, \mathit{fun}\ g^{\circ}\,]\!)\ \mathit{bs}\ x \end{array}$$

(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 *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 *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:

$$
\begin{array}{llll}
\mathit{stream}\,f\,g : (x : X)\,\{h : X \to X\} \to & & & \\
\qquad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\,f))\ h \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (h\ x) & & & \\
\mathit{stream}\,f\,g\,x \quad \mathit{as} & \textbf{with}\ g\ x & \mid \mathit{inspect}\ g\ x & \\
\mathit{stream}\,f\,g\,x\,\{h\}\ \mathit{as} & \mid \mathsf{next}\ b\ x' & \mid [\ \mathit{gxeq}\ ] & = \mathsf{cons}\ b\ (h\ x')\ \{\ \}_0 \\
 & & & \quad (\mathit{stream}\,f\,g\,x'\,\mathit{as}) \\
\mathit{stream}\,f\,g\,x \quad (\mathsf{nil} \qquad \mathsf{refl} \quad) & \mid \mathsf{nothing} & \mid [\ \mathit{gxeq}\ ] & = \mathsf{nil}\ \mathit{gxeq} \\
\mathit{stream}\,f\,g\,x \quad (\mathsf{cons}\ a\ h'\ \mathsf{refl}\ \mathit{as}) & \mid \mathsf{nothing} & \mid [\ \mathit{gxeq}\ ] & = \mathit{stream}\,f\,g\,(f\ x\ a)\ \mathit{as}
\end{array}
$$

We match $g\ x$ with either of the two patterns $\mathsf{next}\ b\ x' = (\text{‘}\mathsf{cons}\ , b\ , x'\ , \blacksquare)$ and $\mathsf{nothing} = (\text{‘}\mathsf{nil}\ , \blacksquare)$.

- 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.
  - 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 $(\mathit{fun}\ g^{\circ})\ \mathsf{nothing}\ (h\ x)$, 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 $h' \circ \mathit{flip}\,f\ a$ where

$$\begin{tikzcd}
x \arrow[r, mapsto, "\text{produce } b \text{ with } g"] \arrow[d, mapsto, "\text{consume } as \text{ with } h"'] & x' \arrow[d, mapsto, "\text{consume } as \text{ with } h"] \\
h\ x \arrow[r, mapsto, "\text{produce } b \text{ with } g"'] & h\ x'
\end{tikzcd}$$

Figure 5.5 State transitions involved in commutativity of production and consumption (cf. Gibbons [2007b, Figures 1 and 2]).

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 $(h' \circ \mathit{flip}\ f\ a)\ x$ are definitionally equal.

Now we look at Goal 0. We have

$$\mathit{gxeq} \ :\ g\ x \ \equiv\ \mathsf{next}\ b\ x'$$

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 5.5): The function $h : X \to X$ 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**:

$$\begin{array}{l}
\mathit{StreamingCondition}\ f\ g\ :\ \mathsf{Set} \\
\mathit{StreamingCondition}\ f\ g\ = \\
\quad (a\ :\ A)\ (b\ :\ B)\ (x\ x'\ :\ X) \rightarrow g\ x\ \equiv\ \mathsf{next}\ b\ x' \rightarrow g\ (f\ x\ a)\ \equiv\ \mathsf{next}\ b\ (f\ x'\ a)
\end{array}$$

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:

$$\begin{array}{l}
\mathit{streaming\text{-}lemma}\ : \\
\quad (b\ :\ B)\ (x\ x'\ :\ X) \rightarrow g\ x\ \equiv\ \mathsf{next}\ b\ x' \rightarrow \\
\quad \{h\ :\ X \rightarrow X\} \rightarrow \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \rightarrow g\ (h\ x)\ \equiv\ \mathsf{next}\ b\ (h\ x') \\
\mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{nil}\qquad \mathsf{refl}\quad)\ =\ \mathit{eq} \\
\mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{cons}\ a\ h\ \mathsf{refl}\ \mathit{as})\ = \\
\quad \mathit{streaming\text{-}lemma}\ b\ (f\ x\ a)\ (f\ x'\ a)\ (\mathit{streaming\text{-}condition}\ f\ g\ a\ b\ x\ x'\ \mathit{eq})\ \mathit{as}
\end{array}$$

where *streaming-condition* : *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 *streaming-lemma* $b\ x\ x'$ *gxeq as*.

We have thus completed the implementation of the Streaming Theorem, except that *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 Greedy Theorem and 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:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Coin}\ :\ \mathsf{Set}\ \mathbf{where}\\
&\quad \mathsf{1p}\ :\ \mathsf{Coin}\\
&\quad \mathsf{2p}\ :\ \mathsf{Coin}\\
&\quad \mathsf{5p}\ :\ \mathsf{Coin}
\end{aligned}
$$

Given $n\ :\ \mathsf{Nat}$, 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:

$$
\begin{aligned}
&\mathit{min}\_\bullet\Lambda\_\ :\ \{I\ :\ \mathsf{Set}\}\ \{X\ Y\ :\ I \to \mathsf{Set}\}\ (R\ :\ Y \rightsquigarrow Y)\ (S\ :\ X \rightsquigarrow Y) \to (X \rightsquigarrow Y)\\
&\mathit{min}\_\bullet\Lambda\_\ \{Y := Y\}\ R\ S\ \{i\}\ x\ y\ =\ S\ x\ y \times ((y'\ :\ Y\ i) \to S\ x\ y' \to R\ y'\ y)
\end{aligned}
$$

An input $x\ :\ X\ i$ for some $i\ :\ I$ is mapped by $\mathit{min}\ R\ \bullet\Lambda\ S$ to $y\ :\ Y\ i$ if y is a possible result in $S\ x\ :\ \mathscr{P}(Y\ i)$ and is the smallest such result under R, in the sense that any y' in $S\ x\ :\ \mathscr{P}(Y\ i)$ must satisfy $R\ y'\ y$. (We think of R as mapping larger inputs to smaller outputs.) Intuitively, we can think of $\mathit{min}\ R\ \bullet\Lambda\ S$ as consisting of two steps: the first step $\Lambda\ S$ computes the set of all possible results yielded by S, and the second step $\mathit{min}\ R$ 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

$$
\begin{aligned}
&\_\leqslant_{\mathsf{C}}\_\ :\ \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set}\\
&c \leqslant_{\mathsf{C}} d\ =\ \mathit{value}\ c \leqslant \mathit{value}\ d
\end{aligned}
$$

where the values of the coins are defined by

$$
\begin{aligned}
&\mathit{value}\ :\ \mathsf{Coin} \to \mathsf{Nat}\\
&\mathit{value}\ \mathsf{1p}\ =\ 1
\end{aligned}
$$

$$\begin{aligned} &\mathit{value}\ \mathsf{2p} \ = \ 2 \\ &\mathit{value}\ \mathsf{5p} \ = \ 5 \end{aligned}$$

then the datatype of coin bags we use is

$$\begin{aligned}
&\mathit{CoinBagOD} \ : \ \mathsf{OrnDesc\ Coin}\ !\ (\mathit{ListD}\ \mathsf{Coin}) \\
&\mathit{CoinBagOD} \ = \ \mathit{OrdListOD}\ \mathsf{Coin}\ (\mathit{flip}\ _\leqslant_{\mathsf{C}}_) \\
&\quad \text{-- specialising } \mathit{Val} \text{ to } \mathsf{Coin} \text{ and } _\leqslant_ \text{ to } \mathit{flip}\ _\leqslant_{\mathsf{C}}_
\end{aligned}$$

$$\begin{aligned}
&\textbf{indexfirst data}\ \mathsf{CoinBag} \ : \ \mathsf{Coin} \to \mathsf{Set}\ \textbf{where} \\
&\quad \mathsf{CoinBag}\ c \ \ni \ \mathsf{nil} \\
&\qquad\qquad\quad \mid \ \mathsf{cons}\ (d \ : \ \mathsf{Coin})\ (\mathit{leq} \ : \ d \leqslant_{\mathsf{C}} c)\ (b \ : \ \mathsf{CoinBag}\ d)
\end{aligned}$$

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 computed by an ornamental forgetful function shrinking ordered lists to natural numbers:

$$\begin{aligned}
&\mathit{size\text{-}alg} \ : \ \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat} \\
&\mathit{size\text{-}alg} \ = \ \mathit{ornAlg}\ (\mathit{NatD\text{-}ListD}\ \mathsf{Coin} \odot \lceil \mathit{CoinBagOD} \rceil)
\end{aligned}$$

$$\begin{aligned}
&\mathit{size} \ : \ \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat} \\
&\mathit{size} \ = \ \mathit{fold}\ \mathit{size\text{-}alg} \\
&\quad \text{-- which is definitionally } \mathit{forget}\ (\mathit{NatD\text{-}ListD}\ \mathsf{Coin} \odot \lceil \mathit{CoinBagOD} \rceil)
\end{aligned}$$

The specification of the minimum coin change problem can now be written as

$$\begin{aligned}
&\mathit{min\text{-}coin\text{-}change} \ : \ \mathit{const}\ \mathsf{Nat} \leadsto \mathsf{CoinBag} \\
&\mathit{min\text{-}coin\text{-}change} \ = \ \mathit{min}\ (\mathit{fun}\ \mathit{size}^{\circ} \bullet \mathit{leq} \bullet \mathit{fun}\ \mathit{size}) \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value}^{\circ})
\end{aligned}$$

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 $R\ :\ \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \leadsto \mathit{const}\ \mathsf{Nat}$,

$$(\!| R |\!)^{\circ}\ \subseteq\ \mathit{min\text{-}coin\text{-}change} \tag{5.6}$$

then we can manufacture a new datatype

$$\begin{aligned}
&\mathit{GreedyBagOD}\ :\ \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathit{outl}\ \lfloor \mathit{CoinBagOD} \rfloor\\
&\mathit{GreedyBagOD}\ =\ \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R
\end{aligned}$$

$$\begin{aligned}
&\mathsf{GreedyBag}\ :\ \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathsf{GreedyBag}\ c\ n\ =\ \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,\ n)
\end{aligned}$$

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 (5.6),

$$\mathit{min\text{-}coin\text{-}change}\ n\ (\mathit{sol}\ n)$$

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 (5.6), 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 a specialisation of Bird and de Moor's Theorem 10.1 [1997] modulo indexing: 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

$$\mathit{min}\ R \bullet \Lambda\ ((\!|\, S \,|\!)^{\circ})$$

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 $(\!|\, S \,|\!)$. The Greedy Theorem states that, if the initial algebra

$$\alpha\ =\ \mathit{fun}\ \mathsf{con}\ :\ \mathbb{F}\ D\ (\mu\ D) \rightsquigarrow \mu\ D$$

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**

$$\alpha \bullet \mathbb{R}\, D\, ((\!|\, S \,|\!)^{\circ}) \bullet (Q \cap (S^{\circ} \bullet S))^{\circ} \subseteq R^{\circ} \bullet \alpha \bullet \mathbb{R}\, D\, ((\!|\, S \,|\!)^{\circ})$$

is satisfied, then we have

$$(\!|\, (\mathit{min}\ Q \bullet \Lambda\, (S^{\circ}))^{\circ} \,|\!)^{\circ} \subseteq \mathit{min}\ R \bullet \Lambda\, ((\!|\, S \,|\!)^{\circ})$$

The Greedy Theorem essentially reduces a global optimisation problem (as indicated by the outermost *min R*) to a local optimisation problem (as indicated by the *min Q* inside the relational fold). 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 $S^{\circ}\ x : \mathscr{P}(\mathbb{F}\ D\ X\ i)$, and assume that xs is better than xs' under Q. Then for any solution resulting from xs' (computed by $\alpha \bullet \mathbb{R}\ D\ ((\!|\, S \,|\!)^{\circ})$) there always exists a better solution resulting from xs, so ignoring xs' would only rule out worse solutions. The greedy condition thus guarantees that we will arrive at an optimal solution by always choosing the best decomposition, which is done by $\mathit{min}\ Q \bullet \Lambda\, (S^{\circ}) : X \rightsquigarrow \mathbb{F}\ D\ X$.

Back to the minimum coin change problem. By *fun-preserves-fold*, the specification *min-coin-change* is equivalent to

$$\mathit{min}\ (\mathit{fun\ size}^{\circ} \bullet \mathit{leq} \bullet \mathit{fun\ size}) \bullet \Lambda\, ((\!|\, \mathit{fun\ total\text{-}value\text{-}alg} \,|\!)^{\circ})$$

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 mono-

```
data CoinOrderedView : Coin → Coin → Set where
  1p1p  : CoinOrderedView 1p 1p
  1p2p  : CoinOrderedView 1p 2p
  1p5p  : CoinOrderedView 1p 5p
  2p2p  : CoinOrderedView 2p 2p
  2p5p  : CoinOrderedView 2p 5p
  5p5p  : CoinOrderedView 5p 5p

view-ordered-coin : (c d : Coin) → c ≤C d → CoinOrderedView c d

data CoinBag′View : {c : Coin} {n : Nat} {l : Nat} → CoinBag′ c n l → Set where
  empty : {c : Coin} → CoinBag′View {c} {0} {0} bnil
  1p1p  : {m l : Nat} {lep : 1p ≤C 1p}
          (b : CoinBag′ 1p m l) → CoinBag′View {1p} {1 + m} {1 + l} (bcons 1p lep b)
  1p2p  : {m l : Nat} {lep : 1p ≤C 2p}
          (b : CoinBag′ 1p m l) → CoinBag′View {2p} {1 + m} {1 + l} (bcons 1p lep b)
  2p2p  : {m l : Nat} {lep : 2p ≤C 2p}
          (b : CoinBag′ 2p m l) → CoinBag′View {2p} {2 + m} {1 + l} (bcons 2p lep b)
  1p5p  : {m l : Nat} {lep : 1p ≤C 5p}
          (b : CoinBag′ 1p m l) → CoinBag′View {5p} {1 + m} {1 + l} (bcons 1p lep b)
  2p5p  : {m l : Nat} {lep : 2p ≤C 5p}
          (b : CoinBag′ 2p m l) → CoinBag′View {5p} {2 + m} {1 + l} (bcons 2p lep b)
  5p5p  : {m l : Nat} {lep : 5p ≤C 5p}
          (b : CoinBag′ 5p m l) → CoinBag′View {5p} {5 + m} {1 + l} (bcons 5p lep b)

view-CoinBag′ : {c : Coin} {n l : Nat} (b : CoinBag′ c n l) → CoinBag′View b
```

Figure 5.6 Two views for proving *greedy-lemma*.

```
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 5.7 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.

tonicity 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

$$\begin{aligned}
&Q : \mathbb{F}\,\lfloor \mathit{CoinBagOD} \rfloor\,(\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathbb{F}\,\lfloor \mathit{CoinBagOD} \rfloor\,(\mathit{const}\ \mathsf{Nat})\\
&Q\,(\text{`}\mathsf{nil}\ \ ,\ \ \blacksquare) \;=\; \mathit{return}\,(\text{`}\mathsf{nil}\,,\blacksquare)\\
&Q\,(\text{`}\mathsf{cons}\,,d\,,\_) \;=\; (\lambda\, e\ \mathit{rest} \mapsto \text{`}\mathsf{cons}\,,e\,,\mathit{rest})\ \langle\$\rangle^2\ (_\leqslant_{\mathsf{C}}_\ d)\ \mathit{any}
\end{aligned}$$

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 nontrivial to prove the greedy condition by relational calculation. Here we offer instead a brute-force yet conveniently expressed case analysis by pattern matching. Define a new datatype $\mathsf{CoinBag}'$ by composing two algebraic ornaments on $\lfloor \mathit{CoinBagOD} \rfloor$ in parallel:

$$\begin{aligned}
&\mathit{CoinBag'OD} : \mathsf{OrnDesc}\,(\mathit{outl} \bowtie \mathit{outl})\ \mathit{pull}\ \lfloor \mathit{CoinBagOD} \rfloor\\
&\mathit{CoinBag'OD} \;=\; \lceil \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{fun\ total\text{-}value\text{-}alg}) \rceil \otimes\\
&\qquad\qquad\qquad\ \lceil \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{fun\ size\text{-}alg}) \rceil
\end{aligned}$$

$$\begin{aligned}
&\mathsf{CoinBag}' : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathsf{CoinBag}' \;=\; \mu\,\lfloor \mathit{CoinBag'OD} \rfloor\,(\mathsf{ok}\,(c\,,n)\,,\mathsf{ok}\,(c\,,l))
\end{aligned}$$

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

$$\mathsf{CoinBag}'\ c\ n\ l \;\cong\; \Sigma[\,b : \mathsf{CoinBag}\ c\,]\ \ (\mathit{total\text{-}value}\ b \;\equiv\; n) \times (\mathit{size}\ b \;\equiv\; l) \tag{5.7}$$

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:

$$\begin{aligned}
&\mathit{greedy\text{-}lemma} : (c\ d : \mathsf{Coin}) \to c \leqslant_{\mathsf{C}} d \to\\
&\qquad\qquad\qquad (m\ n : \mathsf{Nat}) \to \mathit{value}\ c + m \;\equiv\; \mathit{value}\ d + n \to
\end{aligned}$$

$$(l : \mathsf{Nat})\ (b : \mathsf{CoinBag}'\ c\ m\ l) \rightarrow$$
$$\Sigma[\, l' : \mathsf{Nat}\,]\ \mathsf{CoinBag}'\ d\ n\ l' \times (l' \leqslant l)$$

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 (Section 2.2.2) to aid the analysis, whose datatypes and covering functions are shown in Figure 5.6:

- the first view analyses a proof of $c \leqslant_{\mathsf{C}} d$ and exhausts all possibilities of c and d, and
- 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 function *greedy-lemma* can then be split into eight cases by first exhausting all possibilities of c and d by the first view and then analysing the content of b by the second view. Figure 5.7 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 1p and the total value is $1 + n$. The bag must contain 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

$$\begin{array}{l} \mathit{add\text{-}penny} : \\ \quad \{c : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \rightarrow \mathsf{CoinBag}'\ c\ n\ l \rightarrow \mathsf{CoinBag}'\ c\ (1 + n)\ (1 + l) \end{array}$$

and relax the bound of *add-penny b* from 2p to 5p.

The above case analysis may look tedious, but 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 (5.7) to interface with *greedy-lemma*. We conclude that the Greedy Theorem is applicable, and obtain

$$(\!|\, (\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}^{\circ}))^{\circ}\,|\!)^{\circ} \subseteq \mathit{min\text{-}coin\text{-}change}$$

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 CoinBag using the algebra R derived above:

$$\begin{array}{l} \mathit{GreedyBagOD} : \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathit{outl}\ \lfloor \mathit{CoinBagOD} \rfloor \\ \mathit{GreedyBagOD} = \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R \end{array}$$

$$\begin{array}{l} \mathsf{GreedyBag} : \mathsf{Coin} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set} \\ \mathsf{GreedyBag}\ c\ n = \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,\ n) \end{array}$$

whose two constructors can be given the following types:

$$\begin{aligned}
\mathsf{gnil} \; &: \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\} \to \\
&\quad \mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) \equiv n \to \\
&\quad ((ns : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
&\qquad \mathit{total\text{-}value\text{-}alg}\ ns \equiv n \to Q\ ns\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare)) \to \\
&\quad \mathsf{GreedyBag}\ c\ n \\
\mathsf{gcons} \; &: \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\}\ (d : \mathsf{Coin})\ (d{\leqslant}c : d \leqslant_{\mathsf{C}} c) \to \\
&\quad \{n' : \mathsf{Nat}\} \to \mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{cons}\ ,\ d\ ,\ d{\leqslant}c\ ,\ n') \equiv n \to \\
&\quad ((ns : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
&\qquad \mathit{total\text{-}value\text{-}alg}\ ns \equiv n \to Q\ ns\ (\text{‘}\mathsf{cons}\ ,\ d\ ,\ d{\leqslant}c\ ,\ n')) \to \\
&\quad \mathsf{GreedyBag}\ d\ n' \to \mathsf{GreedyBag}\ c\ n
\end{aligned}$$

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 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 $\mathit{total\text{-}value\text{-}alg}\ ns \equiv 0$, 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 $\mathit{total\text{-}value\text{-}alg}\ ns \equiv \mathit{value}\ d + n'$ 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:

$$\begin{array}{l} \mathit{UsableCoin} : \mathsf{Nat} \to \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set} \\ \mathit{UsableCoin}\ n\ c\ d \;=\; (d \leqslant_{\mathsf{C}} c) \times (\Sigma[\, n' : \mathsf{Nat}\,]\ \mathit{value}\ d + n' \equiv n) \end{array}$$

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

$$\begin{array}{ll} \mathsf{gcons}' : & \{c : \mathsf{Coin}\}\ (d : \mathsf{Coin}) \to d \leqslant_{\mathsf{C}} c \to \\ & \{n' : \mathsf{Nat}\} \to \\ & ((e : \mathsf{Coin}) \to \mathit{UsableCoin}\ (\mathit{value}\ d + n')\ c\ e \to e \leqslant_{\mathsf{C}} d) \to \\ & \mathsf{GreedyBag}\ d\ n' \to \mathsf{GreedyBag}\ c\ (\mathit{value}\ d + n') \end{array}$$

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 $n : \mathsf{Nat}$:

$$\begin{array}{l} \mathit{maximum\text{-}coin} : \\ \quad (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to n > 0 \to \\ \quad \Sigma[\, d : \mathsf{Coin}\,]\ \mathit{UsableCoin}\ n\ c\ d \times ((e : \mathsf{Coin}) \to \mathit{UsableCoin}\ n\ c\ e \to e \leqslant_{\mathsf{C}} d) \end{array}$$

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:

$$\begin{array}{l} \mathit{greedy} : (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedyBag}\ c\ n \\ \mathit{greedy}\ c\ n \;=\; \mathit{{<}\text{-}rec}\ P\ f\ n\ c \\ \quad \textbf{where} \\ \qquad P : \mathsf{Nat} \to \mathsf{Set} \\ \qquad P\ n \;=\; (c : \mathsf{Coin}) \to \mathsf{GreedyBag}\ c\ n \end{array}$$

$$
\begin{array}{ll}
f : (n : \mathsf{Nat}) \to ((n' : \mathsf{Nat}) \to n' < n \to P\, n') \to P\, n & \\
f\ n & \mathit{rec}\ c\ \mathbf{with}\ \mathit{compare\text{-}with\text{-}zero}\ n \\
f\ .0 & \mathit{rec}\ c \mid \mathsf{is\text{-}zero} = \mathsf{gnil}' \\
f\ n & \mathit{rec}\ c \mid \mathsf{above\text{-}zero}\ n{>}z\ \mathbf{with}\ \mathit{maximum\text{-}coin}\ c\ n\ n{>}z \\
f\ .(\mathit{value}\ d + n') & \mathit{rec}\ c \mid \mathsf{above\text{-}zero}\ n{>}z \mid d\ ,\ (d{\leqslant}c\ ,\ n'\ ,\ \mathsf{refl})\ ,\ \mathit{guc} = \\
& \qquad\qquad \mathsf{gcons}'\ d\ d{\leqslant}c\ \mathit{guc}\ (\mathit{rec}\ n'\ \{\ \}_8\ d)
\end{array}
$$

In *greedy*, the combinator

$$
\begin{array}{l}
\mathit{<\text{-}rec} : (P : \mathsf{Nat} \to \mathsf{Set}) \to \\
\qquad ((n : \mathsf{Nat}) \to ((n' : \mathsf{Nat}) \to n' < n \to P\, n') \to P\, n) \to \\
\qquad (n : \mathsf{Nat}) \to P\, n
\end{array}
$$

is for well-founded recursion on $\_{<}\_$, and the function

$$\mathit{compare\text{-}with\text{-}zero} : (n : \mathsf{Nat}) \to \mathsf{ZeroView}\ n$$

is a covering function for the view

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{ZeroView} : \mathsf{Nat} \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{is\text{-}zero} \quad\ \ : \mathsf{ZeroView}\ 0 \\
\quad \mathsf{above\text{-}zero} : \{n : \mathsf{Nat}\} \to n > 0 \to \mathsf{ZeroView}\ n
\end{array}
$$

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

Section 5.1 heavily borrows techniques from the AoPA (Algebra of Programming in AGDA) library [Mu et al., 2009] and makes generalisations and adaptations: AoPA deals with non-dependently typed programs only, whereas to work with indexed datatypes we need to move to indexed families of relations; to work with the ornamental universe, we parametrise the relational fold with a description, making it fully datatype-generic, whereas AoPA has only specialised versions for lists and binary trees; we defined $\mathit{min}\_\bullet\Lambda\_$ as a single operator (which happens to be the "shrinking" operator proposed by Mu and Oliveira [2012]) to avoid the struggle with predicativity that AoPA had. All of the above are not fundamental differences between the work presented in this chapter and AoPA, though — the two differ essentially in methodology: in AoPA, dependent types merely provide a foundation on which relational program derivations can be expressed formally and checked by machines, but the programs remain non-dependently typed throughout the formalisation, whereas in this chapter, relational programming is a tool for obtaining non-trivial inductive families that effectively guide program development as advertised as the strength of internalist programming. In short, the focus of AoPA is on traditional relational program derivation (expressed in a dependently typed language), whereas our emphasis is on internalist programming (aided by relational programming).

Algebraic ornamentation was originally proposed by McBride [2011], which deals with functions only. Generalising algebraic ornamentation to a relational setting allows us to write more specifications (like the one given by the Streaming Theorem, which involves converses) and employ more powerful theorems (like the Greedy Theorem, which involves minimisation). We will show in Chapter 6 that this generalisation is in fact a "maximal" one: any datatype obtained via ornamentation can be obtained via relational algebraic ornamentation up to isomorphism. Atkey et al. [2012] also investigated algebraic ornamentation via a fibrational, syntax-free perspective. While their "partial refinement" (which generalises functional algebraic ornamentation) is subsumed by relational algebraic ornamentation (since relational algebras allow partiality), they were able to go beyond inductive families to indexed inductive-recursive datatypes, which are out of the scope of this thesis.

Let us contemplate the interplay between internalist programming and relational programming, especially the one in Section 5.3.3. As mentioned in Section 2.5, internalist programs can encode more semantic information, including correctness proofs; we can thus write programs that directly explain their own meaning. The internalist program *greedy* is such an example, whose structure carries an implicit inductive proof; the program constructs not merely a list of coins, but a bag of coins chosen according to a particular local optimisation strategy (i.e., $\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}^{\circ})$). Internalist programming alone has limited power, however, because internalist programs should share structure with their correctness proofs, but we cannot expect to have such coincidences all the time. In particular, there is no hope of integrating a correctness proof into a program when the structure of the proof is more complicated than that of the program. For example, it is hard to imagine how to integrate a correctness proof for the full specification of the minimum coin change problem into a program for the greedy algorithm. In essence, we have two kinds of proofs to deal with: the first kind follow program structure and can be embedded in internalist programs, and the second kind are general proofs of full specifications, which do not necessarily follow program structure and thus fall outside the scope of internalism. To exploit the power of internalism as much as possible, we need ways to reduce the second kind of proof obligations to the first kind — note that such reduction involves not only constructing proof transformations but also determining what internalist proofs are sufficient for establishing proofs of full specifications. It turns out that relational program derivation is exactly one way in which we can construct such proof transformations systematically from specifications. In relational program derivation, we identify important forms of relational programs (i.e., relational composition, recursion schemes, and various other combinators), and formulate algebraic laws and theorems in terms of these forms. By applying the laws and theorems, we massage a relational specification into a known form which corresponds to a proof obligation that can be expressed in an internalist type, enabling transition to internalist programming. For example, we now know that a relational fold can be turned into an inductive family for internalist programming by algebraic ornamentation. Thus, given a relational specification, we might seek to massage it into a relational fold when that possibility is pointed out by known laws and theorems (e.g., the Greedy Theorem). To sum up, we get a hybrid methodology that leads us from relational specifications towards internalist types for type-directed programming, providing hints in the form of relational algebraic laws and theorems, and this is made possible by going through the synthetic direction of the interconnection between internalism and externalism, synthesising internalist types from relational expressions via algebraic ornamentation.

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 <u>classified</u> 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

$$\mathbb{F}\ \mathit{NatD} : (\top \to \mathsf{Set}) \to (\top \to \mathsf{Set})$$
$$\mathbb{F}\ \mathit{NatD}\ X\ \_ = \Sigma\ \mathit{LTag}\ (\lambda\ \{\mathit{nil-tag} \to \top; \mathit{cons-tag} \to X\ \blacksquare\})$$

And the classifying algebra for the ornament $\mathit{NatD\text{-}ListD}\ A$ is essentially

$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A) : \mathbb{F}\ \mathit{NatD}\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\mathit{nil-tag}\ \ , \_\ \ )\ (\mathsf{ok}\ \blacksquare) = \top$$
$$\mathit{clsAlg}\ (\mathit{NatD\text{-}ListD}\ A)\ (\mathit{cons-tag}\ , \mathsf{ok}\ t)\ (\mathsf{ok}\ \blacksquare) = A \times (t \equiv \blacksquare)$$

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

$$\mathit{clsAlg}\ \mathit{OL} : \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathsf{InvImage}\ !) \rightsquigarrow \mathsf{InvImage}\ !$$
$$\mathit{clsAlg}\ \mathit{OL}\ (\mathit{nil-tag}\ \ , \_\ \ \ \ \ \ )\ (\mathsf{ok}\ b) = \top$$
$$\mathit{clsAlg}\ \mathit{OL}\ (\mathit{cons-tag}\ , a\ , \mathsf{ok}\ b')\ (\mathsf{ok}\ b) = (b \leqslant A\ a) \times (b' \equiv a)$$

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. 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; 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*)

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 133

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 20

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 1

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. ↰ page 18

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 29

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. ↰ page 165

Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff [2007]. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2. doi: 10.1145/1297658.1297660. ↰ page 130

John Backus [1978]. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641. doi: 10.1145/359576.359579. ↰ page 132

Gilles Barthe, Venanzio Capretta, and Olivier Pons [2003]. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293. doi: 10.1017/S0956796802004501. ↰ pages 19 and 105

Jean-Philippe Bernardy [2011]. *A Theory of Parametric Polymorphism and an Application*. Ph.D. thesis, Chalmers University of Technology. ↰ page 99

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 93 and 99

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. ↰ page 30

Richard Bird [1996]. Functional algorithm design. *Science of Computer Programming*, 26(1–3):15–31. doi: 10.1016/0167-6423(95)00033-X. ↰ page 132

Richard Bird [2010]. *Pearls of Functional Algorithm Design*. Cambridge University Press. ↰ page 132

Richard Bird and Oege de Moor [1997]. *Algebra of Programming*. Prentice-Hall. ↰ pages 37, 131, 136, 140, 141, and 155

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 140, 146, and 151

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag. ↰ page 1

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 1

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. ↰ page 22

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 20, 61, and 96

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith [1985]. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall. ↰ page 18

Thierry Coquand and Gérard Huet [1988]. The calculus of constructions. *Information and Computation*, 76(2–3):95–120. doi: 10.1016/0890-5401(88)90005-3. ↰ page 30

Thierry Coquand and Christine Paulin-Mohring [1990]. Inductively defined types. In *International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag. doi: 10.1007/3-540-52335-9_47. ↰ page 30

Pierre-Évariste Dagand and Conor McBride [2012a]. Elaborating inductive definitions. arXiv:1210.6390. ↰ page 27

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 20, 21, 98, 171, and 182

Pierre-Évariste Dagand and Conor McBride [2013]. A categorical treatment of ornaments. To appear in *Logic in Computer Science*. ↰ page 129

Michael Dummett [2000]. *Elements of Intuitionism*. Oxford University Press, second edition. ↰ pages 3 and 14

Peter Dybjer [1994]. Inductive families. *Formal Aspects of Computing*, 6(4):440–465. ↰ page 8

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. ↰ page 23

Nicola Gambino and Joachim Kock [2010]. Polynomial functors and polynomial monads. arXiv:0906.4931. ↰ page 129

Jeremy Gibbons [2007a]. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer-Verlag. doi: 10.1007/978-3-540-76786-2_1. ↰ page 20

Jeremy Gibbons [2007b]. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139. doi: 10.1016/j.scico.2006.01.006. ↰ pages 145 and 150

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 171

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. ↰ page 8

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 105

Arend HEYTING [1971]. *Intuitionism: An Introduction*. Amsterdam: North-Holland Publishing, third revised edition. ↰ page 3

Paul HUDAK, John HUGHES, Simon PEYTON JONES, and Philip WADLER [2007]. A history of Haskell: Being lazy with class. In *History of Programming Languages*, pages 1–55. ACM. doi: 10.1145/1238844.1238856. ↰ page 7

Hsiang-Shang KO and Jeremy GIBBONS [2011]. Modularising inductive families. In *Workshop on Generic Programming*, WGP'11, pages 13–24. ACM. doi: 10.1145/2036918.2036921. ↰ page 31

Hsiang-Shang KO and Jeremy GIBBONS [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi: 10.2201/NiiPi.2013.10.5. ↰ pages 21, 128, and 170

Xavier LEROY [2009]. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446. doi: 10.1007/s10817-009-9155-4. ↰ page 30

Pierre LETOUZEY [2003]. A new extraction for Coq. In *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer-Verlag. doi: 10.1007/3-540-39185-1_12. ↰ page 30

Zhaohui LUO [1994]. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press. ↰ pages 16 and 30

Saunders MAC LANE [1998]. *Categories for the Working Mathematician*. Springer-Verlag, second edition. ↰ pages 103 and 117

Per MARTIN-LÖF [1975]. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier B.V. doi: 10.1016/S0049-237X(08)71945-1. ↰ page 1

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. ↰ page 6

Per MARTIN-LÖF [1984b]. *Intuitionistic Type Theory*. Bibliopolis, Napoli. ↰ pages 1, 19, and 23

Per MARTIN-LÖF [1987]. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420. ↰ page 4

Conor MCBRIDE [1999]. *Dependently Typed Functional Programs and their Proofs*. Ph.D. thesis, University of Edinburgh. ↰ pages 8 and 109

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 1, 9, and 32

Conor MCBRIDE [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ pages 27, 96, 98, 140, 165, 171, and 182

Conor MCBRIDE [2012]. A polynomial testing principle. Available at https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf. ↰ page 33

Conor MCBRIDE and James MCKINNA [2004]. The view from the left. *Journal of Functional Programming*, 14(1):69–111. doi: 10.1017/S0956796803004829. ↰ pages 9, 11, 12, and 13

Conor MCBRIDE and Ross PATERSON [2008]. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13. doi: 10.1017/S0956796807006326. ↰ page 133

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. ↰ pages 7 and 137

Eugenio MOGGI [1991]. Notions of computation and monads. *Information and Computation*, 93(1):55–92. doi: 10.1016/0890-5401(91)90052-4. ↰ page 133

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 59

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 106, 145, and 164

Shin-Cheng MU and José Nuno OLIVEIRA [2012]. Programming from Galois connections. *Journal of Logic and Algebraic Programming*, 81(6):680–704. doi: 10.1016/j.jlap.2012.05.003. ↰ page 165

Keisuke NAKANO [2013]. Metamorphism in jigsaw. *Journal of Functional Programming*, 23(2):161–173. doi: 10.1017/S0956796812000391. ↰ page 146

Bengt NORDSTRÖM [1988]. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619. doi: 10.1007/BF01941137. ↰ page 151

Bengt NORDSTRÖM, Kent PETERSON, and Jan M. SMITH [1990]. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press. ↰ pages 1, 17, 20, and 30

Ulf NORELL [2007]. *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology. ↰ pages 1 and 12

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. ↰ page 1

Chris OKASAKI [1999]. *Purely functional data structures*. Cambridge University Press. ↰ pages 75, 79, 87, and 94

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. ↰ page 30

Simon Peyton Jones [1997]. A new view of guards. Available at http://research.microsoft.com/en-us/um/people/simonpj/Haskell/guards.html. ↰ page 11

John C. Reynolds [2000]. The meaning of types — from intrinsic to extrinsic semantics. Basic Research in Computer Science (BRICS) Report Series RS-00-32, Department of Computer Science, University of Aarhus. ↰ page 32

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. ↰ page 18

Jan M. Smith [1988]. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3):840–845. ↰ page 19

Thomas Streicher [1993]. Investigations into intensional type theory. Habilitation thesis, Ludwig Maximilian Universität. ↰ pages 8 and 16

Wouter Swierstra [2008]. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. doi: 10.1017/S0956796808006758. ↰ pages 96 and 98

The Univalent Foundations Program [2013]. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton. ↰ pages 18 and 19

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. ↰ page 12

Philip Wadler [1989]. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM. ↰ page 99

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 133

Todo list