Datatype ornamentation and the Dutch National Flag problem

Hsiang-Shang Ko

October 27, 2011

In his seminal book *A Discipline of Programming* [3], one of the concluding remarks made by Dijkstra was:

> [...] that it does not suffice to design a mechanism of which we hope that it will meet its requirements, but that we must design it in such a form that we can convince ourselves — and anyone else for that matter — that it will, indeed, meet its requirements. And, therefore, instead of first designing the program and then trying to prove its correctness, we develop correctness proof and program hand in hand. (In actual fact, the correctness proof is developed slightly ahead of the program: after having chosen the form of the correctness proof we make the program so that it satisfies the proof's requirements.)

Dijkstra used the guarded command language for programming and predicate logic for reasoning, relating them by the weakest precondition semantics. The separation of programming language and reasoning language forced him to make the distinction between programs and proofs, and talk indirectly about a program satisfying a proof's requirements. In contrast, in *dependently typed programming*, which is based on Martin-Löf's intuitionistic type theory [7], programming and logic are coherently unified and expressed in one language. (For a brief summary of how intuitionistic type theory unifies programming and logic, see Section 1 of Appendix B.) Since programs and proofs have the same form, it is even possible not to draw a distinct line between programming and reasoning: Ideally, dependently typed programs are not just developed with their correctness proofs in mind — they are written such that they themselves serve as proofs. Dependently typed programming thus has the potential to bring us even closer to program correctness by construction.

In Sections 1 and 2, we briefly survey dependently typed programming in Agda and datatype ornamentation. Section 3 develops a solution to the Dutch National Flag problem, using algebraic ornamentation to reveal and propagate datatype properties. Section 5 steps back, using this development and the reusability problem pointed out in Section 4 to motivate a programme of study in datatype ornamentation. Appendix A contains the final solution to the Dutch National Flag problem, Appendix B an essay on some of the issues encountered when moving from intuitionistic type theory towards dependently typed programming, and Appendix C a paper on modularising inductive families.

1 Dependently typed programming in Agda

There exist many dependently typed programming systems/proof assistants, examples including Agda [11], Epigram [9], Ωmega [13], Coq [2], Isabelle/HOL [10], and Matita [1]. Among them, Agda is designed to be a programming language with full dependent types and Haskell-like syntax, and is widely used by the dependently typed programming community. Unlike proof assistants, in which proofs are constructed using special "tactic" languages different from those in which programs are written, in Agda both programs and proofs (if we wish to distinguish them) look like ordinary functional programs. Agda is thus more appropriate for experimenting with writing programs that are manifestly correct.

Agda data declarations employ the syntax of generalised algebraic datatypes (GADTs), one notable feature being that the types of constructors are explicitly written. For example, the type of natural numbers is defined by:

```
data Nat : Set where
  0 : Nat
  s : Nat → Nat
```

and the type of lists by:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

With this syntax, we can define *inductive families*: The members of an inductive family of types are constructed simultaneously, and can refer to other members when specified inductively. A typical example is vectors, i.e., lists indexed with their length.

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A 0
  _::_ : A → {n : Nat} → Vec A n → Vec A (s n)
```

The first line of the declaration says that Vec A is an inductive family of types indexed by elements of Nat (hence the type Nat → Set). The empty vector [] is an element of the member type at index 0, and when a list xs is an element of the member type at n (and x is of type A), the list $x :: xs$ is an element of the member type at s n. The type of lists is thus partitioned into a family of list types, each member type containing only lists of a particular length. A list xs moves from Vec A 0 to Vec A (*length* xs) as it is constructed; conversely, if xs has type Vec A n, it must be the case that xs is constructed using n cons cells — this is the only way to bring a list to the type Vec A n. In general, having the power of defining inductive families allows us to partition a traditional algebraic datatype into a family of types and specify how the elements move through the family of types when they are constructed, so we can gain knowledge about how an element is constructed by looking at the index of its type.

To define functions on vectors, we can use *dependent pattern matching*. For example, vector append can be defined by:

$$
\begin{array}{l}
\_{+\!\!+}\_ \;:\; \forall\,\{A\ m\ n\} \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (m + n) \\
[\,] \qquad {+\!\!+}\ ys \;=\; ys \\
(x :: xs) {+\!\!+}\ ys \;=\; x :: (xs {+\!\!+}\ ys)
\end{array}
$$

where natural number addition is defined by:

$$
\begin{array}{l}
\_+\_ \;:\; \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{0} \quad + n \;=\; n \\
(\mathsf{s}\ m) + n \;=\; \mathsf{s}\ (m + n)
\end{array}
$$

The vector append program looks exactly like one for list append, except for the more informative type and the more sophisticated unification happening behind the scenes: When the first input vector (of type $\mathsf{Vec}\ A\ m$) is matched with $[\,]$, the variable m is unified with $\mathsf{0}$ and the result type $\mathsf{Vec}\ A\ (\mathsf{0} + n)$ computes to $\mathsf{Vec}\ A\ n$, which is exactly the type of ys. For the cons case, m must be of the form $\mathsf{s}\ m'$ where m' is the index of the type of xs, so the result type is $\mathsf{Vec}\ A\ ((\mathsf{s}\ m') + n)$, which computes to $\mathsf{Vec}\ A\ (\mathsf{s}\ (m' + n))$ and matches the type of $x :: (xs {+\!\!+}\ ys)$. Thus the proof that the length of the output vector is the sum of the lengths of the two input vectors is in effect encoded in the indices and carried out implicitly. (For a more detailed explanation of how dependent pattern matching works, see Section 2 of Appendix B.)

2 Datatype ornamentation

Lists are natural numbers decorated with elements, and vectors are elaborated lists such that the added index in the type of a vector is always the length of the underlying list. This addition of information to a raw datatype to form a fancier datatype is called an *ornamentation*, and was first formulated by McBride using datatype-generic techniques [8]. A particularly useful class of ornamentations is *algebraic ornamentations*, which add an extra index to a datatype such that the index in the type of an element is always the value computed by a particular fold on that element. The vector datatype is a typical example — a vector is a list whose type is indexed by its length, which is computed by a fold. Consider how we might augment the list datatype to get the vector datatype: The type of the empty list $[\,]$ should get $\mathsf{0}$ as its new index because $\mathit{length}\ [\,] = \mathsf{0}$. For cons, whenever we have a list xs whose type is indexed by n — to refer to n we need to insert a field before the recursive node — we can inductively assume that n is the length of xs, and subsequently the type of $x :: xs$ should get the index $\mathsf{s}\ n$ because $\mathit{length}\ (x :: xs) = \mathsf{s}\ (\mathit{length}\ xs) = \mathsf{s}\ n$. The algebraic ornamentation of the list datatype that gives us the vector datatype is thus derived from the definition of *length*, and this derivation can be naturally generalised to work for all datatypes. (For a more detailed albeit condensed introduction to datatype ornamentation, and algebraic ornamentation in particular, see Section 2 of Appendix C.) Algebraic ornamentation will play an important role in the following incremental development of a dependently typed solution to the *Dutch National Flag problem.*

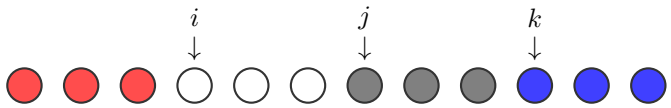


Figure 1: A Dutch vector. Pebbles of unknown colour are shown as grey.

3 The Dutch National Flag problem

Dijkstra introduced the Dutch National Flag problem [3, Chapter 14], asking for a way to rearrange an array of pebbles of colour red, white, or blue in the order of the Dutch National Flag using only swaps. He observed that we can maintain the invariant that the array is divided into four sections containing red, unknown, white, and blue pebbles respectively. In the following solution, however, we adopt the order red, white, unknown, and blue for the four sections (as illustrated in Figure 1), which is symmetric to Dijkstra's version but more convenient for our purpose. Initially the colours of the pebbles are all unknown, and the algorithm proceeds by reducing the size of the unknown section. Traditionally, correctness of such a swapping algorithm is proved by reasoning in terms of array indices (which we will call *pointers* below to avoid confusion with type indices), which can be messy. However, it turns out that the invariant can be formulated inductively and encoded as extra indices of the list datatype, and the proofs that each reduction step maintains the invariant look essentially like ordinary list programs. (See Appendix A for a sneak preview.)

First we define a three-element datatype for the three colours,

```
data Colour : Set where red white blue : Colour
```

and assume that there is a family of types $\mathsf{Peb} : \mathsf{Colour} \to \mathsf{Set}$ where $\mathsf{Peb}\ c$ is the type of pebbles of colour c. In the imperative solution, we maintain the invariant with the help of three pointers i, j, and k for the position immediately after the red, white, and unknown section of the array respectively. (Again see Figure 1.) The invariant has an inductive structure if we regard the array as a list: For example, when we add one more red pebble to the front of the list, the invariant is still satisfied, and the new pointers are $\mathsf{s}\ i$, $\mathsf{s}\ j$, and $\mathsf{s}\ k$. If we use lists whose type is indexed by the three pointers — which we call the *Dutch vectors* — as a representation of the arrays satisfying the invariant, then the observation translates to a "red cons" constructor:

$$\_::_r\_ : \mathsf{Peb\ red} \to \forall\ \{i\ j\ k\} \to \mathsf{DVec}\ i\ j\ k \to \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)$$

To add a white pebble to the front of a Dutch vector, however, first we must ensure that the red section is empty, because a white pebble cannot appear to the left of red pebbles. Fortunately, this prerequisite is easy to check, because saying that the red section is empty is equivalent to saying that the position immediately after the red section, i.e., the pointer i, is zero. After adding the white pebble, both j and k increase by one while i remains zero. The "white cons" constructor is thus:

$$\_::_w\_ : \mathsf{Peb\ white} \to \forall\ \{j\ k\} \to \mathsf{DVec}\ 0\ j\ k \to \mathsf{DVec}\ 0\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)$$

Following the same line of reasoning, (the first version of) the definition of the Dutch vector datatype can be completed. (We will revise the definition several times in response to the needs for more precision later.)

```
data DVec : (i j k : Nat) → Set where
  []      : DVec 0 0 0
  _::r_   : Peb red          →
            ∀ {i j k} → DVec i j k → DVec (s i) (s j) (s k)
  _::w_   : Peb white        →
            ∀ {  j k} → DVec 0 j k → DVec 0     (s j) (s k)
  _::_    : ∀ {c} → Peb c →
            ∀ {    k} → DVec 0 0 k → DVec 0     0     (s k)
  _::b_   : Peb blue         →
                       DVec 0 0 0 → DVec 0     0     0
```

We can convince ourselves that a Dutch vector must satisfy the invariant by merely looking at the indices: For example, if we use a red cons, then it is impossible to use the other three kinds of conses afterwards, because the Dutch vectors they receive must have 0 as the first index, whereas a red cons produces a Dutch vector that has s i as the first index, and 0 is distinct from s i for any i.

We aim to show by actually programming on the Dutch vectors that the unknown section can be reduced. Exactly how to reduce the unknown section depends on the colour of the pebble in the unknown section we choose to look at, a natural choice being the first (leftmost) one*. However, the function that computes the first unknown colour of a Dutch vector is not total, as the unknown section can be empty. A solution is to wrap the output colour in the following version of *Maybe*, which will take the length of the unknown section as its first argument:

```
Maybe : Nat → Set → Set
Maybe 0     A = ⊤
Maybe (s _) A = A
```

where ⊤ is a one-element type whose only constructor is tt. So the first unknown colour can be computed by the function

```
firstUnknownColour :
  ∀ {i j k} → (xs : DVec i j k) → Maybe (lengthUnknown xs) Colour
firstUnknownColour []                = tt
firstUnknownColour (x ::r xs)        = firstUnknownColour xs
firstUnknownColour (x ::w xs)        = firstUnknownColour xs
firstUnknownColour (_::_ {c} x xs)   = c
firstUnknownColour (x ::b xs)        = tt
```

where *lengthUnknown* computes the length of the unknown section:

```
lengthUnknown : ∀ {i j k} → DVec i j k → Nat
lengthUnknown []          = 0
lengthUnknown (x ::r xs)  = lengthUnknown xs
lengthUnknown (x ::w xs)  = lengthUnknown xs
```

*Dijkstra also gave an analysis indicating that such a choice leads to fewer swaps on average.

$$
\begin{aligned}
&\mathit{lengthUnknown}\ (x ::\ \ \mathit{xs}) \ =\ \mathsf{s}\ (\mathit{lengthUnknown}\ \mathit{xs})\\
&\mathit{lengthUnknown}\ (x ::_b\ \mathit{xs}) \ =\ \mathsf{0}
\end{aligned}
$$

We can use algebraic ornamentation to simplify the type of *firstUnknownColour*: Instead of computing the length of the unknown section on the fly, we can expose that length — i.e., the value of *lengthUnknown*, which is a fold — as a fourth index in the type of the Dutch vectors.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{DVec}\ :\ (i\ j\ k\ n\ :\ \mathsf{Nat}) \to \mathsf{Set}\ \textbf{where}\\
&\quad [\,] \quad\ :\ \mathsf{DVec}\ 0\ 0\ 0\ 0\\
&\quad \_::_r\_\ :\ \mathsf{Peb\ red} \qquad\ \to\\
&\qquad\qquad \forall\ \{i\ j\ k\ n\} \to \mathsf{DVec}\ i\ j\ k\ n \to \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n\\
&\quad \_::_w\_\ :\ \mathsf{Peb\ white} \qquad\ \to\\
&\qquad\qquad \forall\ \{\ \ j\ k\ n\} \to \mathsf{DVec}\ 0\ j\ k\ n \to \mathsf{DVec}\ 0 \qquad (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n\\
&\quad \_::\_\ \ :\ \forall\ \{c\} \to \mathsf{Peb}\ c \to\\
&\qquad\qquad \forall\ \{\ \ \ k\ n\} \to \mathsf{DVec}\ 0\ 0\ k\ n \to \mathsf{DVec}\ 0 \qquad 0 \qquad (\mathsf{s}\ k)\ (\mathsf{s}\ n)\\
&\quad \_::_b\_\ :\ \mathsf{Peb\ blue} \qquad\ \to\\
&\qquad\qquad \forall\ \{\ \ \ \ \ n\} \to \mathsf{DVec}\ 0\ 0\ 0\ n \to \mathsf{DVec}\ 0 \qquad 0 \qquad 0 \qquad 0
\end{aligned}
$$

This exposure of the length of the unknown section in the Dutch vector datatype will also be useful later; in particular, the index will serve as an explicit termination measure. The type of *firstUnknownColour* thus becomes:

$$
\begin{aligned}
&\mathit{firstUnknownColour}\ :\\
&\quad \forall\ \{i\ j\ k\ n\} \to (\mathit{xs}\ :\ \mathsf{DVec}\ i\ j\ k\ n) \to \mathit{Maybe}\ n\ \mathsf{Colour}
\end{aligned}
$$

Its definition remains unchanged.

Next we try to define

$$
\begin{aligned}
&\mathit{reduce}\ :\ \forall\ \{i\ j\ k\ n\} \to\\
&\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n) \to \exists^3\ (\lambda\ i'\ j'\ k' \mapsto \mathsf{DVec}\ i'\ j'\ k'\ n)
\end{aligned}
$$

which reduces the unknown section by one pebble (as shown in the type[†]). We need to do a case analysis on the first unknown colour:

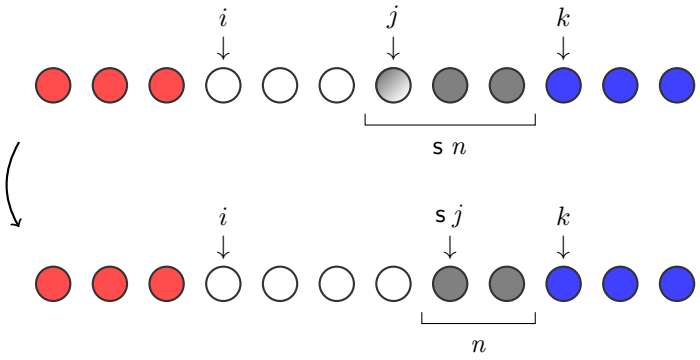
$$
\begin{aligned}
&\mathit{reduce}\ :\ \forall\ \{i\ j\ k\ n\} \to\\
&\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n) \to \exists^3\ (\lambda\ i'\ j'\ k' \mapsto \mathsf{DVec}\ i'\ j'\ k'\ n)\\
&\mathit{reduce}\ \mathit{xs}\ \textbf{with}\ \mathit{firstUnknownColour}\ \mathit{xs}\\
&\mathit{reduce}\ \mathit{xs}\ \mid\ \mathsf{red} \quad = \{\ \}_0\\
&\mathit{reduce}\ \mathit{xs}\ \mid\ \mathsf{white}\ = \{\ \}_1\\
&\mathit{reduce}\ \mathit{xs}\ \mid\ \mathsf{blue} \quad = \{\ \}_2
\end{aligned}
$$

[†]The function $\exists^3$ computes a type from a predicate and is defined by

$$
\begin{aligned}
&\exists^3\ :\ \{A\ :\ \mathsf{Set}\}\ \{B\ :\ A \to \mathsf{Set}\}\ \{C\ :\ (a\ :\ A) \to B\ a \to \mathsf{Set}\} \to\\
&\qquad ((a\ :\ A)\ (b\ :\ B\ a)\ (c\ :\ C\ a\ b) \to \mathsf{Set}) \to \mathsf{Set}\\
&\exists^3\ \{A\}\ \{B\}\ \{C\}\ p\ =\ \Sigma\ (\Sigma\ A\ (\lambda\ a \mapsto \Sigma\ (B\ a)\ (\lambda\ b \mapsto\ C\ a\ b)))\\
&\qquad\qquad\qquad\qquad (\lambda\ abc \mapsto\ p\ (\pi_1\ abc)\ (\pi_1\ (\pi_2\ abc))\ (\pi_2\ (\pi_2\ abc)))
\end{aligned}
$$

where Σ is the dependent pair type former and π_1 and π_2 are the projection functions. We will use a family of such functions, which can be generated by "arity-generic" techniques [14] if one wishes.

(The grey regions are holes left in the program to be filled out, or in different words, programming "goals" to be solved interactively. See Section 2 of Appendix B.) Notice that, since we request a DVec $i\ j\ k$ (s n), we are guaranteed to get a Dutch vector that has a nonempty unknown section, so in particular *firstUnknownColour* returns a proper Colour. Let us consider the white case at goal 1 first, since it is the simplest case.



Since the first pebble in the unknown section — which should be white — is immediately to the right of the white section, we can simply change the unknown cons that adjoins that pebble to the rest of the vector to a white cons. To do so, we might perform a second-level case analysis on *xs*, splitting goal 1 into goals 3–5:

```
reduce : ∀ {i j k n} →
           DVec i j k (s n) → ∃³ (λ i′ j′ k′ ↦ DVec i′ j′ k′ n)
reduce xs with firstUnknownColour xs
reduce xs           | red   = { }₀
reduce (y ::ᵣ ys)   | white = { }₃
reduce (y ::w ys)   | white = { }₄
reduce (y ::  ys)   | white = { }₅
reduce xs           | blue  = { }₂
```

(Notice that we do not need to consider the two cases [] and $y ::_b ys$, since the two constructors cannot possibly deliver Dutch vectors with nonempty unknown sections, i.e., of type DVec $i\ j\ k$ (s n).) At goal 5, we wish to use $y ::_w ys$, thereby replacing the unknown cons with a white cons, but the expression does not typecheck! Inspecting the context, we see that the type of y is still Peb c for some c : Colour — Agda does not know that c must be white. This is reasonable, though, since the case analysis on *firstUnknownColour xs* has nothing to do with the type of *xs*, so Agda does not gain more knowledge about *xs* from the case analysis. What we need to do is expose the first unknown colour as a fifth index of the Dutch vector datatype:

```
data DVec : (i j k n : Nat) → Maybe n Colour → Set where
  []      : DVec 0 0 0 0 tt
  _::ᵣ_   : Peb red          →
```

$$
\begin{array}{lll}
 & \forall\,\{i\ j\ k\ n\ u\} \to \mathsf{DVec}\ i\ j\ k\ n\ u \to \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n & u \\
\_::_w\_ & : \mathsf{Peb}\ \mathsf{white} \to \\
 & \forall\,\{\ \ j\ k\ n\ u\} \to \mathsf{DVec}\ \mathsf{0}\ j\ k\ n\ u \to \mathsf{DVec}\ \mathsf{0}\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n & u \\
\_::\_ & : \forall\,\{c\} \to \mathsf{Peb}\ c \to \\
 & \forall\,\{\ \ \ k\ n\ u\} \to \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ k\ n\ u \to \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ c \\
\_::_b\_ & : \mathsf{Peb}\ \mathsf{blue} \to \\
 & \forall\,\{\ \ \ \ n\ u\} \to \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ n\ u \to \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ \mathsf{tt}
\end{array}
$$

This is the algebraic ornamentation exposing the value of *firstUnknownColour*. Now the case analysis on the first unknown colour can be performed on the fifth index instead,

$$
\begin{array}{l}
\mathit{reduce}\ :\ \forall\,\{i\ j\ k\ n\ u\} \to \\
\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ u \to \exists^4\ (\lambda\ i'\ j'\ k'\ u' \mapsto \mathsf{DVec}\ i'\ j'\ k'\ n\ u') \\
\mathit{reduce}\ \{u = \mathsf{red}\ \ \}\ xs\ \ \ \ \ \ \ \ = \{\ \}_0 \\
\mathit{reduce}\ \{u = \mathsf{white}\}\ (y ::_r\ \ ys)\ = \{\ \}_3 \\
\mathit{reduce}\ \{u = \mathsf{white}\}\ (y ::_w\ ys)\ = \{\ \}_4 \\
\mathit{reduce}\ \{u = \mathsf{white}\}\ (y ::\ \ \ ys)\ = \{\ \}_5 \\
\mathit{reduce}\ \{u = \mathsf{blue}\ \ \}\ xs\ \ \ \ \ \ \ \ = \{\ \}_2
\end{array}
$$

and at goal 5 the type of y is $\mathsf{Peb}\ \mathsf{white}$, so the goal can be solved by $(\_,\ y ::_w ys)$. Goal 4 presents another problem, however: We wish to use $y ::_w \pi_2\ (\mathit{reduce}\ ys)$, i.e., skipping the white cons and proceeding with the rest of the vector, but to be entitled to write the white cons, the first pointer in the type of $\pi_2\ (\mathit{reduce}\ ys)$ has to remain $\mathsf{0}$, which is not guaranteed by the type of *reduce*. Therefore we need to be more specific about the pointer indices of the output vector by writing a separate *reduceWhite* function (which corresponds to strengthening the induction hypothesis).

$$
\begin{array}{l}
\mathit{reduceWhite}\ :\ \forall\,\{i\ j\ k\ n\} \to \\
\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ \mathsf{white} \to \exists\ (\lambda\ u' \mapsto \mathsf{DVec}\ i\ (\mathsf{s}\ j)\ k\ n\ u') \\
\mathit{reduceWhite}\ (y ::_r\ ys)\ =\ \_,\ y ::_r\ \pi_2\ (\mathit{reduceWhite}\ ys) \\
\mathit{reduceWhite}\ (y ::_w\ ys)\ =\ \_,\ y ::_w\ \pi_2\ (\mathit{reduceWhite}\ ys) \\
\mathit{reduceWhite}\ (y ::\ \ ys)\ =\ \_,\ y ::_w\ ys
\end{array}
$$

The type of *reduceWhite* is informative enough to tell us that, if the three pointers are i, j, and k originally and the first unknown colour is in fact white, then the white section can be expanded by one pebble (hence the new pointer $\mathsf{s}\ j$) and the length of the unknown section decreases by one. The program says that a way to achieve this is to skip red and white conses and replace the first unknown cons with a white cons. Now in the white case of *reduce*, we simply delegate the task to *reduceWhite* (instead of performing a second-level case analysis on xs).

$$
\begin{array}{l}
\mathit{reduce}\ :\ \forall\,\{i\ j\ k\ n\ u\} \to \\
\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ u \to \exists^4\ (\lambda\ i'\ j'\ k'\ u' \mapsto \mathsf{DVec}\ i'\ j'\ k'\ n\ u') \\
\mathit{reduce}\ \{u = \mathsf{red}\ \ \}\ xs\ =\ \{\ \}_0 \\
\mathit{reduce}\ \{u = \mathsf{white}\}\ xs\ =\ \_,\ \pi_2\ (\mathit{reduceWhite}\ xs) \\
\mathit{reduce}\ \{u = \mathsf{blue}\ \ \}\ xs\ =\ \{\ \}_2
\end{array}
$$

Similarly we will define *reduceRed* and *reduceBlue* for the other two cases.

Remark. We are forced to write expressions like

$$\_,\ y ::_r \pi_2\ (\mathit{reduceWhite}\ ys)$$

instead of simply

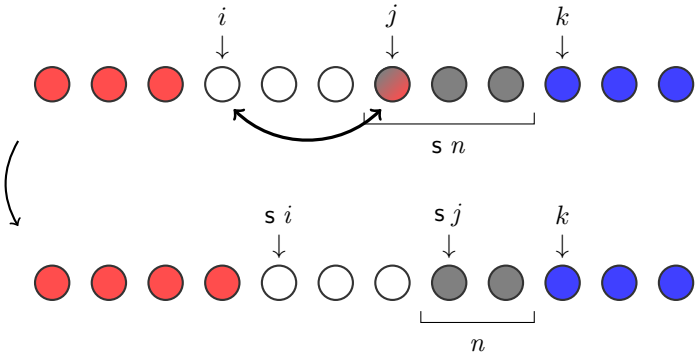
$$y ::_r \mathit{reduceWhite}\ ys$$

because *reduceWhite* ys is a dependent pair — some of the indices of the returned Dutch vector are existentially quantified and need to be paired with the vector. We thus have to throw away the indices, produce the desired vector and again pair it with its indices (which Agda can infer). This causes some syntactic noise, which does not seem possible to be removed with the current implementation of Agda, however. (*End of remark.*)

The red case is slightly more complex but still straightforward. The type of *reduceRed* is:

$$\begin{aligned}&\mathit{reduceRed}\ :\ \forall\ \{i\ j\ k\ n\} \to \\ &\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ \mathsf{red} \to \exists\ (\lambda\ u' \mapsto \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ k\ n\ u')\end{aligned}$$

because by swapping the first pebble in the unknown section — which is now known to be red — with the first white pebble, the red section would be expanded by one pebble (hence the new pointer $\mathsf{s}\ i$) and, at the same time, the white section would in effect be shifted to the right by one position (hence the new pointer $\mathsf{s}\ j$).



We start by performing case analysis on the input vector.

$$\begin{aligned}&\mathit{reduceRed}\ :\ \forall\ \{i\ j\ k\ n\} \to \\ &\qquad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ \mathsf{red} \to \exists\ (\lambda\ u' \mapsto \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ k\ n\ u') \\ &\mathit{reduceRed}\ (y ::_r\ ys) = \{\ \}_0 \\ &\mathit{reduceRed}\ (y ::_w\ ys) = \{\ \}_1 \\ &\mathit{reduceRed}\ (y ::\ \ ys) = \{\ \}_2\end{aligned}$$

We skip all red conses, so goal 0 is solved by $(\_,\ y ::_r \pi_2\ (\mathit{reduceRed}\ ys))$. At goal 2, the vector starts with the unknown section, meaning that the red and white sections are empty, so we can simply replace the unknown cons with a red cons, solving the goal by $(\_,\ y ::_r\ ys)$. Goal 1 is the interesting case, at

which we encounter the first pebble y of the white section, which we need to swap with the red pebble at the beginning of the unknown section, and then we replace the white cons with a red cons. We retrieve the red pebble to be swapped with y by the function

$$
\begin{aligned}
&\mathit{focus} \;:\; \forall\,\{j\ k\ n\} \to \mathsf{DVec}\ 0\ j\ k\ (\mathsf{s}\ n)\ \mathsf{red} \to \mathsf{Peb}\ \mathsf{red}\\
&\mathit{focus}\ (z ::_w zs) \;=\; \mathit{focus}\ zs\\
&\mathit{focus}\ (z ::\ \ zs) \;=\; z
\end{aligned}
$$

and substitute y for the red one by the function

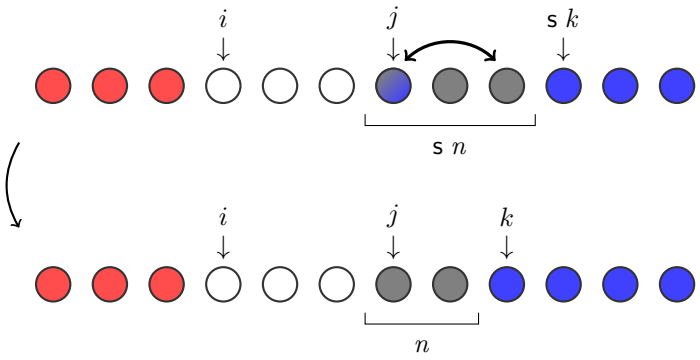
$$
\begin{aligned}
&\mathit{subst} \;:\; \forall\,\{j\ k\ n\} \to \mathsf{Peb}\ \mathsf{white} \to\\
&\qquad \mathsf{DVec}\ 0\ j\ k\ (\mathsf{s}\ n)\ \mathsf{red} \to \exists\,(\lambda\ u' \mapsto \mathsf{DVec}\ 0\ (\mathsf{s}\ j)\ k\ n\ u')\\
&\mathit{subst}\ y\ (z ::_w zs) \;=\; \_,\ z ::_w \pi_2\ (\mathit{subst}\ y\ zs)\\
&\mathit{subst}\ y\ (z ::\ \ zs) \;=\; \_,\ y ::_w zs
\end{aligned}
$$

Now goal 1 can be solved by $(\_,\ \mathit{focus}\ ys ::_r \pi_2\ (\mathit{subst}\ y\ ys))$.

The blue case is the trickiest, in particular requiring another algebraic ornamentation of the Dutch vector datatype. The type of *reduceBlue* is:

$$
\begin{aligned}
&\mathit{reduceBlue} \;:\; \forall\,\{i\ j\ k\ n\} \to\\
&\qquad \mathsf{DVec}\ i\ j\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ \mathsf{blue} \to \exists\,(\lambda\ u' \mapsto \mathsf{DVec}\ i\ j\ k\ n\ u')
\end{aligned}
$$

The third pointer decreases from $\mathsf{s}\ k$ to k because the blue section grows towards the head of the vector by swapping the first pebble of the unknown section — which is known to be blue — with the last one.



If we perform case analysis on the input vector and try to solve the goals,

$$
\begin{aligned}
&\mathit{reduceBlue} \;:\; \forall\,\{i\ j\ k\ n\} \to\\
&\qquad \mathsf{DVec}\ i\ j\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ \mathsf{blue} \to \exists\,(\lambda\ u' \mapsto \mathsf{DVec}\ i\ j\ k\ n\ u')\\
&\mathit{reduceBlue}\ (y ::_r\ \ ys) \;=\; \{\ \}_0\\
&\mathit{reduceBlue}\ (y ::_w\ ys) \;=\; \{\ \}_1\\
&\mathit{reduceBlue}\ (y ::\ \ \ ys) \;=\; \{\ \}_2
\end{aligned}
$$

we would soon encounter some difficulties: At goals 0 and 1, we wish to skip the red and white conses as before, but the expression *reduceBlue ys* does not typecheck, because the third pointer in the type of *ys* is some arbitrary k whereas

reduceBlue requires that to be a successor. Nevertheless, we do know that k must be a successor, because it is bounded below by $\mathsf{s}\ n$. And at goal 2, we need to do a second-level case analysis on n and simply replace the unknown cons with a blue cons when n is $\mathsf{0}$, i.e., when y is the only pebble in the unknown section. However, the expression $y ::_b ys$ does not typecheck either, because the blue cons requires the third pointer in the type of ys to be $\mathsf{0}$, which is again just some arbitrary k. Nevertheless, we do know that k has to be $\mathsf{0}$, because when n is $\mathsf{0}$ it must be equal to j, which is $\mathsf{0}$ in this case. These difficulties suggest that we did not express the connection between the three indices j, k, and n precisely enough in the Dutch vector datatype — we need to say explicitly that n is exactly $k - j$. A possibility is to use the following datatype to state that the difference between k and j is n:

$$
\begin{array}{l}
\textbf{data}\ \_-\_\approx\_ : (k\ j\ n : \mathsf{Nat}) \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{0} : \forall\ \{j\} \to j - j \approx \mathsf{0} \\
\quad \mathsf{s} : \forall\ \{k\ j\ n\} \to k - j \approx n \to \mathsf{s}\ k - j \approx \mathsf{s}\ n
\end{array}
$$

We can write a function computing such a difference:

$$
\begin{array}{lcl}
\mathit{difference} : \forall\ \{i\ j\ k\ n\ u\} \to \mathsf{DVec}\ i\ j\ k\ n\ u \to k - j \approx n & & \\
\mathit{difference}\ [\,] & = & \mathsf{0} \\
\mathit{difference}\ (x ::_r\ xs) & = & \mathit{inj}\ (\mathit{difference}\ xs) \\
\mathit{difference}\ (x ::_w\ xs) & = & \mathit{inj}\ (\mathit{difference}\ xs) \\
\mathit{difference}\ (x ::\ \ xs) & = & \mathsf{s}\ (\mathit{difference}\ xs) \\
\mathit{difference}\ (x ::_b\ xs) & = & \mathsf{0}
\end{array}
$$

where *inj* says that the difference is unchanged when both j and k are increased by one:

$$
\begin{array}{lcl}
\mathit{inj} : \forall\ \{k\ j\ n\} \to k - j \approx n \to \mathsf{s}\ k - \mathsf{s}\ j \approx n & & \\
\mathit{inj}\ \mathsf{0} & = & \mathsf{0} \\
\mathit{inj}\ (\mathsf{s}\ d) & = & \mathsf{s}\ (\mathit{inj}\ d)
\end{array}
$$

The function *difference* is a fold, so again we can algebraically ornament the Dutch vector datatype to expose its value.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{DVec} : (i\ j\ k\ n : \mathsf{Nat}) \to \\
\qquad\qquad k - j \approx n \to \mathit{Maybe}\ n\ \mathsf{Colour} \to \mathsf{Set}\ \textbf{where} \\
\quad [\,] \quad\quad : \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ \mathsf{tt} \\
\quad \_::_r\_ : \mathsf{Peb}\ \mathsf{red} \quad\quad\ \to \forall\ \{i\ j\ k\ n\ d\ u\} \to \\
\qquad\quad \mathsf{DVec}\ i\ j\ k\ n\ d\ u \to \mathsf{DVec}\ (\mathsf{s}\ i)\ (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n \quad\quad (\mathit{inj}\ d)\ u \\
\quad \_::_w\_ : \mathsf{Peb}\ \mathsf{white} \quad\quad\ \to \forall\ \{\ \ j\ k\ n\ d\ u\} \to \\
\qquad\quad \mathsf{DVec}\ \mathsf{0}\ j\ k\ n\ d\ u \to \mathsf{DVec}\ \mathsf{0} \quad\quad (\mathsf{s}\ j)\ (\mathsf{s}\ k)\ n \quad\quad (\mathit{inj}\ d)\ u \\
\quad \_::\_ \quad : \forall\ \{c\} \to \mathsf{Peb}\ c \to \forall\ \{\quad k\ n\ d\ u\} \to \\
\qquad\quad \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ k\ n\ d\ u \to \mathsf{DVec}\ \mathsf{0} \quad\quad \mathsf{0} \quad\quad (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ (\mathsf{s}\ d) \quad\quad c \\
\quad \_::_b\_ : \mathsf{Peb}\ \mathsf{blue} \quad\quad\ \to \forall\ \{\quad\ \ n\ d\ u\} \to \\
\qquad\quad \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ \mathsf{0}\ n\ d\ u \to \mathsf{DVec}\ \mathsf{0} \quad\quad \mathsf{0} \quad\quad \mathsf{0} \quad\quad \mathsf{0} \quad\quad \mathsf{0} \quad\quad\ \mathsf{tt}
\end{array}
$$

(This is the final version of the Dutch vector datatype.) Back to *reduceBlue*, now we have more type information to exploit.

$$
\begin{array}{l}
\mathit{reduceBlue} : \forall\ \{i\ j\ k\ n\ d\} \to \\
\quad \mathsf{DVec}\ i\ j\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ d\ \mathsf{blue} \to \exists^2\ (\lambda\ d'\ u' \mapsto \mathsf{DVec}\ i\ j\ k\ n\ d'\ u')
\end{array}
$$

$$
\begin{aligned}
&\mathit{reduceBlue}\ (y ::_r\ \ ys) &&= \{\ \}_0\\
&\mathit{reduceBlue}\ (y ::_w\ ys) &&= \{\ \}_1\\
&\mathit{reduceBlue}\ (y ::\ \ ys) &&= \{\ \}_2
\end{aligned}
$$

At goals 0 and 1, we remind Agda that the index d in the type of ys can only be of the form $(\mathsf{s}\ \_)$ by pattern matching. As a result, k must be a successor, and we are cleared to call *reduceBlue ys*.

$$
\begin{aligned}
&\mathit{reduceBlue}\ :\ \forall\ \{i\ j\ k\ n\ d\} \rightarrow\\
&\quad \mathsf{DVec}\ i\ j\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ d\ \mathsf{blue} \rightarrow \exists^2\ (\lambda\ d'\ u' \mapsto \mathsf{DVec}\ i\ j\ k\ n\ d'\ u')\\
&\mathit{reduceBlue}\ (\_::_r\_\ y\ \{d = \mathsf{s}\ \_\}\ ys) &&= \_,\ y ::_r\ \pi_2\ (\mathit{reduceBlue}\ ys)\\
&\mathit{reduceBlue}\ (\_::_w\_\ y\ \{d = \mathsf{s}\ \_\}\ ys) &&= \_,\ y ::_w\ \pi_2\ (\mathit{reduceBlue}\ ys)\\
&\mathit{reduceBlue}\ (y :: ys) &&= \{\ \}_2
\end{aligned}
$$

At goal 2, we replace the unknown cons with a blue cons if and only if y is the only pebble in the unknown section; otherwise we swap the last pebble in the unknown section with y and adjoin y to the blue section using a blue cons. Asking whether y is the only pebble in the unknown section is equivalent to asking whether the unknown section in ys is empty, and this question can be answered by pattern matching on the index d in the type of ys again.

$$
\begin{aligned}
&\mathit{reduceBlue}\ :\ \forall\ \{i\ j\ k\ n\ d\} \rightarrow\\
&\quad \mathsf{DVec}\ i\ j\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ d\ \mathsf{blue} \rightarrow \exists^2\ (\lambda\ d'\ u' \mapsto \mathsf{DVec}\ i\ j\ k\ n\ d'\ u')\\
&\mathit{reduceBlue}\ (\_::_r\_\ y\ \{d = \mathsf{s}\ \_\}\ ys) &&= \_,\ y ::_r\ \pi_2\ (\mathit{reduceBlue}\ ys)\\
&\mathit{reduceBlue}\ (\_::_w\_\ y\ \{d = \mathsf{s}\ \_\}\ ys) &&= \_,\ y ::_w\ \pi_2\ (\mathit{reduceBlue}\ ys)\\
&\mathit{reduceBlue}\ (\_::\_\ \ y\ \{d = \mathsf{0}\ \ \}\ ys) &&= \{\ \}_3\\
&\mathit{reduceBlue}\ (\_::\_\ \ y\ \{d = \mathsf{s}\ \_\}\ ys) &&= \{\ \}_4
\end{aligned}
$$

Goal 3 can be solved by $(\_,\ y ::_b\ ys)$. For goal 4, we need to retrieve the last pebble of the unknown section by the function

$$
\begin{aligned}
&\mathit{focus}\ :\ \forall\ \{k\ n\ d\ u\} \rightarrow \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ k\ (\mathsf{s}\ n)\ d\ u \rightarrow \exists\ (\lambda\ u' \mapsto \mathsf{Peb}\ u')\\
&\mathit{focus}\ \{n = \mathsf{0}\ \ \}\ (z :: zs) &&= \_,\ z\\
&\mathit{focus}\ \{n = \mathsf{s}\ \_\}\ (z :: zs) &&= \mathit{focus}\ zs
\end{aligned}
$$

and then use the following function to substitute y for the last pebble and also adjoin it to the blue section:

$$
\begin{aligned}
&\mathit{subst}\ :\ \forall\ \{k\ n\ d\ u\} \rightarrow \mathsf{Peb}\ \mathsf{blue} \rightarrow\\
&\qquad \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ (\mathsf{s}\ k)\ (\mathsf{s}\ n)\ (\mathsf{s}\ d)\ u \rightarrow \exists\ (\lambda\ u' \mapsto \mathsf{DVec}\ \mathsf{0}\ \mathsf{0}\ k\ n\ d\ u')\\
&\mathit{subst}\ y\ (\_::\_\ z\ \{d = \mathsf{0}\ \ \}\ zs) &&= \_,\ y ::_b\ zs\\
&\mathit{subst}\ y\ (\_::\_\ z\ \{d = \mathsf{s}\ \_\}\ zs) &&= \_,\ z :: \pi_2\ (\mathit{subst}\ y\ zs)
\end{aligned}
$$

Now goal 4 can be solved by $(\_,\ \pi_2\ (\mathit{focus}\ ys) :: \pi_2\ (\mathit{subst}\ y\ ys))$.

Having gone through the three cases, the *reduce* function can now be completed.

$$
\begin{aligned}
&\mathit{reduce}\ :\ \forall\ \{i\ j\ k\ n\ d\ u\} \rightarrow\\
&\quad \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ d\ u \rightarrow \exists^5\ (\lambda\ i'\ j'\ k'\ d'\ u' \mapsto \mathsf{DVec}\ i'\ j'\ k'\ n\ d'\ u')\\
&\mathit{reduce}\qquad\quad \{u = \mathsf{red}\ \ \}\ xs\ =\ \_,\ \pi_2\ (\mathit{reduceRed}\ \ \ xs)
\end{aligned}
$$

```
reduce              {u = white} xs  =  _ , π₂ (reduceWhite xs)
reduce {d = s _} {u = blue  } xs  =  _ , π₂ (reduceBlue  xs)
```

In the blue case, again we need to remind Agda that k must be a successor by pattern matching on the index d in the type of xs. The main program calls *reduce* repeatedly until n decreases to 0, so it is an induction on n (and thus obviously terminates).

```
elimUnknown : ∀ {i j k n d u} →
  DVec i j k n d u → ∃⁴ (λ i′ j′ k′ d′ ↦ DVec i′ j′ k′ 0 d′ tt)
elimUnknown {n = 0  } xs  =  _ , xs
elimUnknown {n = s _} xs  =  elimUnknown (π₂ (reduce xs))
```

Since every list of pebbles can be cast as a Dutch vector by the function

```
initialise : (xs : List (Σ Colour Peb)) →
             let l = length xs in ∃ (λ u ↦ DVec 0 0 l l fuel u)
initialise []             =  tt , []
initialise ((c , x) :: xs)  =  c , x :: π₂ (initialise xs)
```

where

```
fuel : ∀ {k} → k - 0 ≈ k
fuel {0  }  =  0
fuel {s _}  =  s fuel
```

and we have an obvious forgetful function from the Dutch vectors to lists of pebbles,

```
forget : ∀ {i j k n d u} → DVec i j k n d u → List (Σ Colour Peb)
forget []           =  []
forget (x ::ᵣ xs)   =  (red   , x) :: forget xs
forget (x ::ᵥ xs)   =  (white , x) :: forget xs
forget (x ::  xs)   =  (_     , x) :: forget xs
forget (x ::ᵦ xs)   =  (blue  , x) :: forget xs
```

the composite function

```
dutchFlag  :  List (Σ Colour Peb) → List (Σ Colour Peb)
dutchFlag  =  forget ∘ π₂ ∘ elimUnknown ∘ π₂ ∘ initialise
```

solves the Dutch National Flag problem. It should be pointed out that the solution was developed incrementally — several times we discovered that more precision was needed to complete the program and refined the Dutch vector datatype accordingly: Inspired by the invariant shown in Figure 1, we started with a Dutch vector datatype indexed by the three pointers, and subsequently we used algebraic ornamentation to expose the length n of the unknown section (to serve as an explicit argument on which we can do induction), the first unknown colour (to perform case analysis and decide how to reduce the unknown section), and the proof that n is exactly the difference between the third pointer k and the second pointer j (to give Agda more information about the indices in the blue case). This is a development pattern we should strive to support. The final version of the solution is listed in Appendix A.

4 Internalism and externalism

The programming style of the above solution to the Dutch National Flag problem might be characterised as *internalist*, suggesting that constraints are internalised in datatypes and proofs are manipulated simultaneously with data. A more traditional style is the *externalist* one, in which we formulate properties and write proofs about existing programs separately without modifying the existing programs. For example, to solve the Dutch National Flag problem in the externalist way, we might write *dutchFlag* directly as an ordinary program on $\mathsf{List}\ (\Sigma\ \mathsf{Colour}\ \mathsf{Peb})$, formulate a predicate

$$\mathit{IsDutch}\ :\ \mathsf{List}\ (\Sigma\ \mathsf{Colour}\ \mathsf{Peb}) \to \mathsf{Set}$$

asserting that a list of pebbles is arranged as required, and then prove

$$\begin{array}{l}\mathit{dutchFlag\text{-}correctness}\ : \\ \quad (\mathit{xs}\ :\ \mathsf{List}\ (\Sigma\ \mathsf{Colour}\ \mathsf{Peb})) \to \mathit{IsDutch}\ (\mathit{dutchFlag}\ \mathit{xs})\end{array}$$

with respect to the definitions of *dutchFlag* and *IsDutch*. The two styles serve different purposes in the setting of correct-by-construction dependently typed programming: Internalist programs aim to be manifestly correct by incorporating the essential properties into datatypes and programs, while externalist proofs can show that existing programs satisfy additional properties in a modular fashion. For example, we wrote the internalist solution to the Dutch National Flag problem with the invariant in mind, but afterwards we might also wish to confirm that every reduction step only swaps the pebbles instead of tampering with them in some way. Instead of redesigning the datatypes and programs to show the property, which would be a large amount of work and, worse, complicate the logic, it is much cleaner and easier to define a separate predicate

$$\mathit{Swap}\ :\ \{A\ :\ \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{Set}$$

on plain lists such that $\mathit{Swap}\ \mathit{xs}\ \mathit{ys}$ means ys is one swap away from xs, and prove:

$$\begin{array}{l}\mathit{reduce\text{-}swapping}\ :\ \forall\ \{i\ j\ k\ n\ d\ u\} \to \\ \quad (\mathit{xs}\ :\ \mathsf{DVec}\ i\ j\ k\ (\mathsf{s}\ n)\ d\ u) \to \mathit{Swap}\ (\mathit{forget}\ \mathit{xs}) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{forget}\ (\pi_2\ (\mathit{reduce}\ \mathit{xs})))\end{array}$$

The Dutch National Flag problem is thus a good example showing that both internalism and externalism are indispensable. (For more discussion about the two styles, see Section 3 of Appendix B.)

There is a great advantage of externalism over internalism, however, which is the high composability of externalist structures. In externalism, since programs carry only minimal type information, they can be used in as many contexts as possible, and externalist proofs can be easily imposed on programs in a non-intrusive way (using dependent pairs) when needed. In contrast, internalist programs — especially those with too specific types — are harder to reuse, although they are often more concise than their externalist counterparts since proofs require no separate management. Recently we published a paper Modularising Inductive Families [6] for the Workshop on Generic Programming (attached as Appendix C) reporting an initial attempt to combine internalist clarity with

externalist composability. We identified a family of isomorphisms between internalist datatypes and simpler datatypes paired with externalist predicates, which can help to structure libraries of internalist datatypes in a modular way. The WGP paper is merely a starting point — there is still much to be investigated about reusability and modularity of internalist programs, which is a main issue that has to be resolved if dependently typed programming is to succeed.

5 Thesis proposal

The thesis will aim to show that dependently typed programs have *composable structures* that allow *modular library design* and *incremental program development*. The ability to structure programs in a modular fashion is essential to reusability of library code, as useful components can be separately maintained and later freely combined when needed, and the ability to develop programs incrementally enables programmers to focus on only one aspect of their programs at a time and later integrate all the work together. In the dependently typed setting, where proofs are manipulated along with programs, libraries keep not only data structures and algorithms but also proofs about their various properties, and programmers write not only programs but also their various correctness proofs. Dependently typed programmers thus should be able to request internalist data structures and algorithms carrying correctness proofs customised for their needs from libraries as easily as combining externalist programs and proofs, and move towards more precisely typed programs incrementally, adding only the necessary information at each step. To achieve such goals, a study of composable structures of dependently typed programs is necessary. I will aim to study a particular approach based on McBride's datatype ornamentation [8], giving it both an intensional, type-theoretic implementation and an extensional, category-theoretic characterisation. I will propose ways to structure internalist libraries modularly and give examples of reusable components that can go into such libraries, and explore mechanisms that help to upgrade programs towards more precise types. A possible title for the thesis might be *Composable structures for dependently typed programming.*

Chapter outline

1. *Introduction.* The introduction will argue that programs should be correct by construction, echoing Dijkstra [3]. Dependently typed programming is one promising way to such a goal. However, existing dependently typed libraries are rigid and have dreadful reusability, and currently dependently typed program development is largely monolithic and fails to achieve separation of concerns. Both observations point to a study of the composable structure of dependently typed programs, into which there has been no systematic investigation.

2. *Dependently typed programming.* This chapter summarises the state of the art in dependently typed programming, which might include foundational theories, current implementations (mainly Agda, which will be the language for carrying out experiments), common idioms, and illustrative examples (like the Dutch National Flag problem).

3. *Datatype ornamentation.* The content of the WGP paper [6] (after completion and revision) might go into this chapter, including a revised definition of ornaments, ornamental-algebraic ornamentation, and ornament fusion (which was called ornament composition in the WGP paper).

4. *Canonical upgrade.* Intuitively, it should be effortless to upgrade list append to vector append, since in a language that supports implicit arguments (like Agda) the two programs look exactly the same, and we know that the implicitly added proof about length has "the same structure" as that of the list append program. Similar things can be said about upgrading natural number addition to list append. Also during our development of the internalist solution to the Dutch National Flag problem, we ornamented the Dutch vector datatype several times without explicitly revisiting the programs previously written. The types of those programs in fact need to be updated manually, but the program texts remain the same. This is a hint that we should be able to derive programs on an ornamented Dutch vector datatype automatically from those on a simpler Dutch vector datatype. In general, this will be about exploiting ornamental information to generate "canonical" programs on fancier types from programs having simpler types. The work in the previous chapter might be described as reusing and extending datatypes, while this chapter will be about reusing and extending program structures.

5. *A category-theoretic characterisation.* There is a category **Desc** with datatype descriptions as objects and ornaments as arrows. It can be translated to the category Fam(**Set**) by the contravariant functor which maps a description to the decoded family of sets and an ornament to the induced forgetful map. There is also a contravariant functor **Desc** $\to$ **Set** mapping a description to its index set and an ornament to its index erasure function.

$$\begin{array}{ccccc} I & & D & & \mu\, D \\ \uparrow{\scriptstyle e} & \leftsquigarrow & \downarrow{\scriptstyle O} & \rightsquigarrow & \uparrow{\scriptstyle forget\ O} \\ J & & E & & \mu\, E \end{array}$$

Ornament fusion is a pushout[‡]: It contains the least information that covers both O_1 and O_2. Correspondingly, the index set of $\lfloor O_1 \oplus O_2 \rfloor$ is a pullback, as already pointed out in the WGP paper.

$$\begin{array}{ccc} I & \xleftarrow{e_2} & J_2 \\ {\scriptstyle e_1}\uparrow & & \uparrow{\scriptstyle \pi_2} \\ J_1 & \xleftarrow[\pi_1]{} & e_1 \otimes e_2 \end{array} \qquad \leftsquigarrow \qquad \begin{array}{ccc} D & \xrightarrow{O_2} & E_2 \\ {\scriptstyle O_1}\downarrow & & \downarrow \\ E_1 & \longrightarrow & \lfloor O_1 \oplus O_2 \rfloor \end{array}$$

Such a pushout in **Desc** is a coproduct in the coslice category $D \downarrow$ **Desc**, and we have observed in the WGP paper that ornament fusion corresponds

[‡]In fact, ornament fusion in its present form is not yet a pushout. But it can be made a genuine pushout if the ornament language is suitably expanded.

to pointwise conjunction of realisability predicates, i.e., a product in some suitable category. Hence it is reasonable to expect that we can find a contravariant functor from $D \downarrow \mathbf{Desc}$ to a (presumably cartesian closed) category of predicates. These preliminary observations show that the theory of ornaments potentially have interesting structures that can be concisely explained by a fibrational analysis [4].

6. *Case studies.* I will experiment with designing a modular library of ornaments and proofs by reexamining and reimplementing data structures and algorithms in, e.g., Okasaki's book [12] and existing Agda libraries. A potential library structure was proposed in the WGP paper. As for incremental program development, an interesting case study might involve migrating the proof of the Church-Rosser property for untyped λ-calculus (via parallel reduction) to one for simply typed λ-calculus à la Church (i.e., the λ-terms are typed). It has been observed [5, p 80] that the proof for untyped λ-calculus can be "adapted fairly easily" to work for simply typed λ-calculus à la Church. Formally, how much do we need to fill in to get a proof for simply typed λ-calculus from one for untyped λ-calculus? This can serve as an evaluation of the degree of reusability achieved.

7. *Discussion.* Summary, overall discussion, future work, etc.

Research plan

The next step is to complete and revise the work in the WGP paper, i.e., adjust the definition of ornaments so it is more amenable to a categorical treatment and finish the implementation of ornament fusion. I will also start porting data structures and algorithms to their dependently typed versions and try to structure them as reusable components as proposed by the WGP paper. With enough examples at hand, it will be easier to tackle the canonical upgrade problem by generalising the patterns observed. And then I will try to do more ambitious case studies like proving the Church-Rosser property. In parallel with the type-theoretic path, I will familiarise myself with fibred category theory and begin the categorical investigation of ornaments at a later time (say in the third year).

References

[1] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, August 2007.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[4] Bart Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1999.

[5] Andrew D. Ker. Lambda calculus and types. Lecture notes, 2009.

[6] Hsiang-Shang Ko and Jeremy Gibbons. Modularising inductive families. In *Workshop on Generic Programming*, WGP'11, September 2011. Attached as Appendix C.

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

[8] Conor McBride. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. Draft available at `http://personal.cis.strath.ac.uk/~conor/pub/OAAO/LitOrn.pdf`.

[9] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming (AFP'04)*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170, 2004.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[11] Ulf Norell. Dependently typed programming in Agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming (AFP'08)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

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

[13] Tim Sheard and Nathan Linger. Programming in Ωmega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *Second Central-European Functional Programming School (CEFP'07)*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer-Verlag, 2007.

[14] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In Jean-Christophe Filliâtre and Cormac Flanagan, editors, *Programming Languages meets Program Verification*, PLPV'10, pages 15–26. ACM, 2010.

Appendix A

The final version of the internalist solution
to the Dutch National Flag problem

```
data Colour : Set where red white blue : Colour

postulate Peb : Colour → Set

Maybe : Nat → Set → Set
Maybe 0     A = ⊤
Maybe (s _) A = A

data _−_≈_ : Nat → Nat → Nat → Set where
  0 : ∀ {j} → j − j ≈ 0
  s : ∀ {k j n} → k − j ≈ n → s k − j ≈ s n

inj : ∀ {k j n} → k − j ≈ n → s k − s j ≈ n
inj 0     = 0
inj (s d) = s (inj d)

fuel : ∀ {k} → k − 0 ≈ k
fuel {0  } = 0
fuel {s _} = s fuel

data DVec : (i j k n : Nat) → k − j ≈ n → Maybe n Colour → Set where
  []     : DVec 0 0 0 0 0 tt
  _::r_  : Peb red         → ∀ {i j k n d u} →
           DVec i j k n d u → DVec (s i) (s j) (s k) n     (inj d) u
  _::w_  : Peb white       → ∀ {  j k n d u} →
           DVec 0 j k n d u → DVec 0     (s j) (s k) n     (inj d) u
  _::_   : ∀ {c} → Peb c → ∀ {    k n d u} →
           DVec 0 0 k n d u → DVec 0     0     (s k) (s n) (s d)   c
  _::b_  : Peb blue        → ∀ {       n d u} →
           DVec 0 0 0 n d u → DVec 0     0     0     0     0       tt

reduceWhite : ∀ {i j k n d} →
              DVec i j k (s n) d white → ∃² (λ d' u' ↦ DVec i (s j) k n d' u')
reduceWhite (y ::r ys) = _, y ::r π₂ (reduceWhite ys)
reduceWhite (y ::w ys) = _, y ::w π₂ (reduceWhite ys)
reduceWhite (y ::  ys) = _, y ::w ys

reduceRed : ∀ {i j k n d} →
            DVec i j k (s n) d red → ∃² (λ d' u' ↦ DVec (s i) (s j) k n d' u')
reduceRed (y ::r ys) = _, y ::r π₂ (reduceRed ys)
reduceRed (y ::w ys) = _, focus ys ::r π₂ (subst y ys)
  where focus : ∀ {j k n d} → DVec 0 j k (s n) d red → Peb red
        focus (z ::w zs) = focus zs
        focus (z ::  zs) = z
        subst : ∀ {j k n d} → Peb white →
                DVec 0 j k (s n) d red → ∃² (λ d' u' ↦ DVec 0 (s j) k n d' u')
        subst y (z ::w zs) = _, z ::w π₂ (subst y zs)
        subst y (z ::  zs) = _, y ::w zs
reduceRed (y ::  ys) = _, y ::r ys
```

```
reduceBlue : ∀ {i j k n d} →
               DVec i j (s k) (s n) d blue → ∃² (λ d' u' ↦ DVec i j k n d' u')
reduceBlue (_::ᵣ_ y {d = s _} ys) = _ , y ::ᵣ π₂ (reduceBlue ys)
reduceBlue (_::w_ y {d = s _} ys) = _ , y ::w π₂ (reduceBlue ys)
reduceBlue (_::_  y {d = 0  } ys) = _ , y ::b ys
reduceBlue (_::_  y {d = s _} ys) = _ , π₂ (focus ys) :: π₂ (subst y ys)
  where focus : ∀ {k n d u} → DVec 0 0 k (s n) d u → ∃ (λ u' ↦ Peb u')
        focus {n = 0  } (z :: zs) = _ , z
        focus {n = s _} (z :: zs) = focus zs
        subst : ∀ {k n d u} → Peb blue →
                  DVec 0 0 (s k) (s n) (s d) u → ∃ (λ u' ↦ DVec 0 0 k n d u')
        subst y (_::_ z {d = 0  } zs) = _ , y ::b zs
        subst y (_::_ z {d = s _} zs) = _ , z :: π₂ (subst y zs)

reduce : ∀ {i j k n d u} →
         DVec i j k (s n) d u → ∃⁵ (λ i' j' k' d' u' ↦ DVec i' j' k' n d' u')
reduce           {u = red  } xs = _ , π₂ (reduceRed   xs)
reduce           {u = white} xs = _ , π₂ (reduceWhite xs)
reduce {d = s _} {u = blue } xs = _ , π₂ (reduceBlue  xs)

elimUnknown : ∀ {i j k n d u} →
                DVec i j k n d u → ∃⁴ (λ i' j' k' d' ↦ DVec i' j' k' 0 d' tt)
elimUnknown {n = 0  } xs = _ , xs
elimUnknown {n = s _} xs = elimUnknown (π₂ (reduce xs))

initialise : (xs : List (Σ Colour Peb)) →
             let l = length xs in ∃ (λ u ↦ DVec 0 0 l l fuel u)
initialise []             = tt , []
initialise ((c , x) :: xs) = c , x :: π₂ (initialise xs)

forget : ∀ {i j k n d u} → DVec i j k n d u → List (Σ Colour Peb)
forget []           = []
forget (x ::ᵣ xs) = (red   , x) :: forget xs
forget (x ::w xs) = (white , x) :: forget xs
forget (x ::  xs) = (_     , x) :: forget xs
forget (x ::b xs) = (blue  , x) :: forget xs

dutchFlag : List (Σ Colour Peb) → List (Σ Colour Peb)
dutchFlag = forget ∘ π₂ ∘ elimUnknown ∘ π₂ ∘ initialise
```

Appendix B

From intuitionistic type theory to dependently typed programming

This essay was submitted as part of a reading course undertaken in my first year.

From intuitionistic type theory to dependently typed programming

Josh Ko

July 30, 2011

Contents

1 Notion of computation in type theory

Mathematics is all about mental constructions, that is, the intuitive grasp and manipulation of mental objects, the intuitionists say [5, 7]. Take the natural numbers as an example. We have a distinct idea of how natural numbers are built: Start from an origin 0, and form its successor 1, and then the successor of 1, which is 2, and so on. In other words, it is in our nature to be able to count, and counting is just the way the natural numbers are constructed. This construction then gives a specification of when we can immediately recognise a natural number, namely when it is 0 or a successor of some other natural number, and this specification of immediately recognisable forms is one of the conditions of forming the *set* of the natural numbers in Martin-Löf's intuitionistic type theory [12, 17]. Expressed in the style of Gentzen's natural deduction system, we are justified by our intuition to have the *formation rule*

$$\frac{}{\mathbb{N} : \mathsf{Set}}$$

which says we can conclude (below the line) that $\mathbb{N}$ is a set from no assumptions (above the line), and the two *introduction rules*

$$\frac{}{\mathsf{zero} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{suc}\, n : \mathbb{N}}$$

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

What is more characteristic of intuitionism is that the intuitionistic interpretation of propositions, and in particular the logical constants, follows the same line of thought as the formation of the set of natural numbers. A proposition is an expression of its truth condition, and since intuitionistic truth follows from proofs, a proposition is clearly specified if and only if what constitutes a proof of it is determined [13]. What is a proof of a proposition, then? It is a piece of mental construction such that, upon inspection, the truth of the proposition is immediately recognised. For a simple example, in type theory we can formulate the formation rule for disjunctions

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

and the introduction rules

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

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

One notices that the notion of effective method, or computation, was presumed when the notion of set was introduced, and at some point we need to concretely specify an effective method. Since the description of every set includes an effective way to construct its canonical elements, it is possible to express an effective method that mimics the construction of an element by saying that the computation has the same shape as how the element is constructed. Again let us look at the natural numbers. Suppose we have a *family of sets* $P : \mathbb{N} \to \mathsf{Set}$ indexed by elements of $\mathbb{N}$. The elements of $\mathbb{N}$ are used as names for these sets, and $P\, n$ denotes the set referred to by the name $n : \mathbb{N}$. If we have an element z of $P\, \mathsf{zero}$ and a method s that, for any $n : \mathbb{N}$, transforms an element of $P\, n$ to an element of $P\, (\mathsf{suc}\, n)$, then we can compute an element of $P\, n$ for any given n by essentially the same counting process with which we construct n, but the counting now starts from z instead of zero and proceeds with s instead of suc. For instance, if a proof of $P\, 2$ is required, we can simply apply s to z twice, just like we apply suc to zero twice to form 2, so the computation was guided by the

shape of 2. This explanation justifies the following *elimination rule*

$$\frac{P : \mathbb{N} \to \mathsf{Set} \qquad z : P\,\mathsf{zero} \qquad s : (n : \mathbb{N}) \to P\,n \to P\,(\mathsf{suc}\,n) \qquad n : \mathbb{N}}{\mathsf{elim}\mathbb{N}\,P\,z\,s\,n : P\,n}$$

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

$$\frac{P : \mathbb{N} \to \mathsf{Set} \qquad z : P\,\mathsf{zero} \qquad s : (n : \mathbb{N}) \to P\,n \to P\,(\mathsf{suc}\,n)}{\mathsf{elim}\mathbb{N}\,P\,z\,s\,\mathsf{zero} = z \in P\,\mathsf{zero}}$$

$$\frac{P : \mathbb{N} \to \mathsf{Set} \qquad z : P\,\mathsf{zero} \qquad s : (n : \mathbb{N}) \to P\,n \to P\,(\mathsf{suc}\,n) \qquad n : \mathbb{N}}{\mathsf{elim}\mathbb{N}\,P\,z\,s\,(\mathsf{suc}\,n) = s\,n\,(\mathsf{elim}\mathbb{N}\,P\,z\,s\,n) \in P\,(\mathsf{suc}\,n)}$$

In general, judgemental equality is extended to a congruence relation, so substitutions of equal subterms can be done freely, in particular replacing computations with their results. (More on equality in Section 4.) Note that we only specify how $\mathsf{elim}\mathbb{N}$ computes when it is applied to zero or $\mathsf{suc}\,n$, i.e., the canonical elements, because we have assumed that we can compute a canonical form for each noncanonical element.

We have specified the set of natural numbers by stating its

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

The central roles in type theory are played by various sets specified in this manner, corresponding to the various mental constructions we play with in mathematics. Martin-Löf himself noted: "If programming is understood [...] as the design of the methods of computation [...], then it no longer seems possible to distinguish the discipline of programming from constructive mathematics" [11]. Indeed, sets are easily comparable with algebraic datatypes, which also play the central roles in functional programming — the formation rule names the datatype, the introduction rules list its constructors, and the elimination rule and computation rules define the fold function. One can give concrete, computational explanations for all the entities appearing in type theory, as we have done for the natural numbers, so type theory serves as a suitable foundation for intuitionistic mathematics, which equates mathematical activities with mental constructions.

For the programming side, type theory reveals a new possibility by incorporating logical entities, i.e., propositions and proofs, into the computational

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

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

2 Elimination vs. pattern matching

The formation rule and the introduction rules for a set directly translate into an algebraic datatype declaration in functional languages. For example, the type of natural numbers is translated into Agda [18] (which will be our expository dependently typed language in this essay) as

$$\begin{array}{l}\textbf{data}\ \mathbb{N} : \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{zero} : \mathbb{N}\\ \quad \mathsf{suc}\ \ : \mathbb{N} \to \mathbb{N}\ .\end{array}$$

Having a datatype, naturally we wish to write programs on that datatype. In functional programming, the pattern matching syntax is widely used for defining programs. It is key to the clarity of functional programs because it not only allows a function to be intuitively defined by several equations but also clearly conveys the strategy of splitting a problem into subproblems by case analysis. On the other hand, computations in type theory are specified using eliminators. Besides keeping the basic theory simple, one reason is that programs in type theory are demanded to be total, for a program must terminate if it is intended as a proof, and using eliminators enforces totality. Pattern matching and elimination are basically equivalent in expressive power, as eliminators can be easily defined by dependent pattern matching, and conversely dependent pattern matching can be reduced to elimination if uniqueness of identity proofs —

also known as the K rule [21] — is assumed [6]. Nevertheless, the use of pattern matching together with an interactive development environment is more informative and helpful in dependently typed languages than in simply typed ones, because splitting a problem into subproblems by case analysis in dependently typed programming often leads to nontrivial refinement of the goal type and even the context.

To illustrate, let us look at an example of interactive development in Agda, whose design was inspired by McBride and McKinna [16]. Consider the following inductively defined less-than-or-equal-to binary relation on natural numbers.

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

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

$$\mathit{trans} : (x\, y\, z : \mathbb{N}) \to x \leq y \to y \leq z \to x \leq z$$

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

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

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

$$\begin{array}{l}\mathit{trans} : (x\, y\, z : \mathbb{N}) \to x \leq y \to y \leq z \to x \leq z\\ \mathit{trans}\ x\ .z\ z\ \ \ p\ \mathsf{refl}\ \ \ = \{\, x \leq z \parallel p : x \leq z \text{ in context} \,\}_1\\ \mathit{trans}\ x\ y\ \ .(\mathsf{suc}\, w)\ p\ (\mathsf{step}\, w\, r) = \{\, x \leq \mathsf{suc}\, w \,\}_2\end{array}$$

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

$$\begin{array}{l}\mathit{trans} : (x\, y\, z : \mathbb{N}) \to x \leq y \to y \leq z \to x \leq z\\ \mathit{trans}\ x\ .z\ z\ \ \ p\ \mathsf{refl}\ \ \ = p\\ \mathit{trans}\ x\ y\ \ .(\mathsf{suc}\, w)\ p\ (\mathsf{step}\, w\, r) = \{\, x \leq \mathsf{suc}\, w \,\}_2\end{array}$$

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

$$\begin{array}{l}\mathit{trans} : (x\, y\, z : \mathbb{N}) \to x \leq y \to y \leq z \to x \leq z\\ \mathit{trans}\ x\ .z\ z\ \ \ p\ \mathsf{refl}\ \ \ = p\\ \mathit{trans}\ x\ y\ \ .(\mathsf{suc}\, w)\ p\ (\mathsf{step}\, w\, r) = \mathsf{step}\, w\ \{\, x \leq w \,\}_3\end{array}$$

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

$$
\begin{array}{l}
\mathit{trans} : (x\ y\ z : \mathbb{N}) \to x \leq y \to y \leq z \to x \leq z \\
\mathit{trans}\ x\ .z\ z \qquad p\ \mathsf{refl} \qquad = p \\
\mathit{trans}\ x\ y\ \ .(\mathsf{suc}\ w)\ p\ (\mathsf{step}\ w\ r) = \mathsf{step}\ w\ (\mathit{trans}\ x\ y\ w\ p\ r)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{elim}\text{-}\leq : (m : \mathbb{N})\ (P : (n : \mathbb{N}) \to m \leq n \to \mathsf{Set}) \to \\
\qquad\qquad ((t : m \leq m) \to P\ m\ t) \to \\
\qquad\qquad ((n : \mathbb{N})\ (t : m \leq n) \to P\ n\ t \to P\ (\mathsf{suc}\ n)\ (\mathsf{step}\ n\ t)) \to \\
\qquad\qquad\quad (n : \mathbb{N})\ (t : m \leq n) \to P\ n\ t
\end{array}
$$

and write

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

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. Comparing the two versions, we see that elimination is inherently harder to write and understand, especially when complicated dependent types are involved. If a function definition requires more than one level of elimination, then the advantage of using pattern matching over using eliminators becomes even more apparent.

It is often the case that we need to perform pattern matching not only on an argument but also on some intermediate computation. In simply typed languages this is usually achieved by case expressions, a special case being if-then-else expressions for booleans. But again, pattern matching on intermediate computation can make refinements to the goal type and the context in dependently typed languages, so case expressions, being more like eliminators, become less desirable. McBride and McKinna [16] thus proposed **with**-*matching*, which generalises pattern guards and in effect shifts pattern matching on intermediate computation from the right of an equation to the left, sitting along with the arguments. For a plain example:

$$
\begin{array}{l}
\mathit{insert} : \mathbb{N} \to \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N} \\
\mathit{insert}\ y\ [] \qquad = [y] \\
\mathit{insert}\ y\ (x :: xs)\ \mathbf{with}\ y \leq_?\ x \\
\mathit{insert}\ y\ (x :: xs)\ |\ \mathsf{true}\ = y :: x :: xs \\
\mathit{insert}\ y\ (x :: xs)\ |\ \mathsf{false} = x :: \mathit{insert}\ y\ xs
\end{array}
$$

This is essentially no different from a normal case expression, except that using **with** renders the result of $y \leq_?\ x$ as an additional argument in the context, which is then immediately matched with true or false. In this case, the original context — y, x, and xs — is not affected by the pattern matching, but in more

interesting cases it can be. For example, Wadler's views [22] can be adapted to dependently typed programming in a more accurate manner, which are supported by **with** in Agda. Suppose we wish to implement a snoc-list view for cons-lists. We define the following view type

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

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

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

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

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

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

McBride and McKinna described how programs using pattern matching can be translated into eliminator-based programs [16]. They in fact proposed a general mechanism for invoking any programmer-defined eliminator using the pattern matching syntax, so programmers can choose whichever problem-splitting strategy they need and express that with pattern matching. For example, the standard eliminator for $\mathbb{N}$ says that to solve a programming problem $P\,n$ for any $n : \mathbb{N}$, it is sufficient to solve the more specialised subproblems $P\ \mathsf{zero}$ and $P\ (\mathsf{suc}\ n)$ (assuming an answer of $P\,n$). This is not the only way to cover all natural numbers, of course; for example, we might split the problem into the two subproblems $P\ i$ where $i < k$ and $P\ (j+k)$ where $j : \mathbb{N}$, for some fixed k. We should be able to match a natural number against such nonstandard patterns if that is the strategy we use to divide and solve the problem. Problem specifications can be made more precise by using dependent types, but the solutions would have to be equally precise as a result. Reintroducing pattern matching into dependently typed languages is one step towards helping programmers to describe such solutions naturally and clearly.

3 Datatype externalism vs. 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 the *dependent pair* type.

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

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

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

where $\mathit{length} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathbb{N}$ computes the length of a list and $\_\equiv\_$ is the propositional equality type (see Section 4). This Σ-type can be naturally read as "the lists xs such that the length of xs is n", bearing some similarity to the notation of set comprehension. Another example is natural numbers bounded above by a certain number. We define a predicate

$$\begin{array}{l}\textbf{data}\ \_>\_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{base} : \{m : \mathbb{N}\} \to \mathsf{suc}\ m > \mathsf{zero} \\ \quad \mathsf{step} : \{m\ n : \mathbb{N}\} \to m > n \to \mathsf{suc}\ m > \mathsf{suc}\ n\end{array}$$

and use the type

$$\Sigma\ \mathbb{N}\ (\lambda n \mapsto m > n)$$

to characterise those natural numbers bounded above by m. Besides being deeply rooted in mathematical traditions, in practice this approach offers very good compositionality: Whenever a new property is needed, the programmer simply defines a new predicate and uses a Σ-type to impose that predicate on an existing datatype. Predicates easily compose by pointwise conjunction, so objects with two or more properties can be conveniently specified. When programs are the objects we reason about, this style naturally suggests a logical distinction between programs and proofs: Programs are written in the first place, and proofs are conducted afterwards with reference to existing programs and do not interfere with their execution. Consequently, proofs may be erased as they are irrelevant to the computational behaviour of programs. This conception underlies many developments in type theory and theorem proving. For example, Luo consistently argued in [10] that proofs should not be identified with programs, one of the reasons being that logic should be regarded as independent from the objects being reasoned about. A subset theory was described by Nordström et al. [17] to suppress the second component, i.e., the proof part, of Σ-types. The proof assistant Coq [3] is also designed to support this proving-after-programming style, which is also famous for supporting program extraction from proof scripts [19].

On the other hand, proponents of dependently typed programming believe that, instead of regarding dependent types as yet another type system we impose

on existing programs, we should rethink about what programs can be written in a dependently typed language. One such reconsideration is the movement of using inductive families directly for representing data with constraints. The classic example is vectors, which are lists indexed by their length.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ (A : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\
\quad [] \quad\ : \mathsf{Vec}\ A\ \mathsf{zero} \\
\quad \_::\_\ : A \to \forall\ \{n\} \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n)
\end{array}
$$

A simple inductive argument shows that a vector of type $\mathsf{Vec}\ A\ n$ must be of length n. This fact holds for a vector *by construction*, in contrast to the previous approach using Σ-types, where the length statement is made about a plain list already constructed. In epistemology, a distinction is made between *internalism* and *externalism*: An internalist insists that a subject must have justification for a belief in order to call it knowledge, whereas an externalist admits a belief as knowledge as long as there is justification for it, even when the justification is not available to the subject. Since inductive families encode constraints in themselves, we might characterise the way of programming which models data with constraints using inductive families as *internalist*, suggesting data "know" their own correctness by construction; the traditional approach, in contrast, may be described as *externalist*, as proofs are constructed externally to the objects which they talk about.

To illustrate why it can be beneficial to switch from externalism to internalism, suppose we wish to extract the head element from a nonempty list. In more traditional functional languages like Haskell, the best we can do is to write

$$
\begin{array}{l}
\mathit{head} : [a] \to a \\
\mathit{head}\ (x :: \mathit{xs}) = x \\
\mathit{head}\ [] \qquad\ = \mathit{error}\ \texttt{"head: empty list"}\ .
\end{array}
$$

The type system cannot preclude the possibility that the input list is empty, so we had better deal with it, but the only thing we can do in the empty-list case is reporting an error. In dependently typed languages, however, we can require that the input list must be nonempty. The externalists would impose a length constraint on the input list and write

$$
\begin{array}{l}
\mathit{head} : \{A : \mathsf{Set}\} \to (\mathit{xs} : \mathsf{List}\ A) \to (l : \mathit{length}\ \mathit{xs} > \mathsf{zero}) \to A \\
\mathit{head}\ [] \qquad () \\
\mathit{head}\ (x :: \mathit{xs})\ \_ \ = x\ .
\end{array}
$$

We do pattern matching on the input list: If it is empty, then the type of l becomes $\mathsf{zero} > \mathsf{zero}$, which cannot be unified with the type of either of the two constructors. Thus in this case the proof l cannot possibly be given in the first place if the program is well typed, which is indicated by the absurd pattern (), and we can omit the definition for this case. If the input list is nonempty, then we can deliver the head element; the proof is irrelevant in this case. The internalists would use vectors and write

$$
\begin{array}{l}
\mathit{vhead} : \{A : \mathsf{Set}\} \to \forall\ \{n\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A \\
\mathit{vhead}\ (x :: \mathit{xs}) = x\ .
\end{array}
$$

We do pattern matching on the input vector: This time, however, the nil case is impossible since $\mathsf{suc}\ n$, the index of the type of the input vector, cannot be

unified with zero, which is the index of the type of the constructor []. This case is thus (safely) omitted. The remaining case is cons, and again we can easily deliver the head element. A more general example is safe lookup which extracts from a list the element at a particular index. Externalists would use natural numbers as indices and write

$$
\begin{array}{l}
\mathit{lookup} : \{A : \mathsf{Set}\} \to (\mathit{xs} : \mathsf{List}\ A) \to (i : \mathbb{N}) \to \mathit{length}\ \mathit{xs} > i \to A \\
\mathit{lookup}\ []\ \ \ \ \ \ \_\ \ \ \ \ ()\ \ \ \ \ \ \ \\
\mathit{lookup}\ (x :: \mathit{xs})\ \mathsf{zero}\ \ \ \ \_\ \ \ \ \ \ = x \\
\mathit{lookup}\ (x :: \mathit{xs})\ (\mathsf{suc}\ i)\ (\mathsf{step}\ l) = \mathit{lookup}\ \mathit{xs}\ i\ l\ .
\end{array}
$$

Again proofs need to be manipulated explicitly. Internalists would first define the *finite numbers* to represent the indices.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Fin} : \mathbb{N} \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathsf{zero}\ : \{m : \mathbb{N}\} \to \mathsf{Fin}\ (\mathsf{suc}\ m) \\
\quad \mathsf{suc}\ \ : \{m : \mathbb{N}\} \to \mathsf{Fin}\ m \to \mathsf{Fin}\ (\mathsf{suc}\ m)
\end{array}
$$

A finite number of type $\mathsf{Fin}\ m$ is a natural number bounded above by m. The lookup function would then be defined on vectors.

$$
\begin{array}{l}
\mathit{vlookup} : \{A : \mathsf{Set}\} \to \forall\ \{n\} \to \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
\mathit{vlookup}\ (x :: \mathit{xs})\ \mathsf{zero}\ \ \ \ = x \\
\mathit{vlookup}\ (x :: \mathit{xs})\ (\mathsf{suc}\ i) = \mathit{vlookup}\ \mathit{xs}\ i
\end{array}
$$

We first perform pattern matching on the index, which points out that n is non-zero and thus the vector is nonempty, so next we only need to match the vector with cons. In the suc case where we need to make the recursive call, the indices in the type of xs and i match perfectly, so the recursive call can be made as if we were writing a simply typed version. We see that, by exploiting inductive families, correctness proofs are built into and manipulated simultaneously with the data, allowing constraints to be expressed succinctly, and in ideal cases like *vlookup*, programs can be written in blissful ignorance of the proofs. Of course, this is possible because the program we write and the associated proof have essentially the same structure; otherwise it would be more difficult to write the program in the internalist way.

Perhaps not surprisingly, internalist reasoning is rarely seen in mathematics. Here are two possible explanations. The first, philosophical one is that the platonist character of classical mathematics, i.e., the presupposition that mathematical objects are independently existing entities, naturally leads to externalism. The mathematical objects exist *a priori*, and then our proofs are written about them. There is thus a clear "phase distinction," which makes it strange to mix proofs with objects. The second, practical explanation (which also works for non-platonist mathematics) is that it is hard to justify the correctness of an internalist program in prose without silently converting the internalist program to an externalist one. For example, we would say "a vector of length n" and go on about how its length relates to the result, etc., which is not so different from saying "a *list* of length n" and so on — we still need to talk and reason about the constraints separately, unlike how we write an internalist program, which only manipulates data. It might be said that the correctness proof of an internalist program is more syntactic in nature, which in general is

more suitable for being checked by machines, while mathematical writing aims to describe the intuition behind so human readers can get the high-level ideas. Even if correctness is implied by the syntactic structure, it is still desirable to have an intuitive explanation of why it is so in prose. Therefore, as the same degree of explanation is needed no matter whether constraints are integrated into syntax or not, it is reasonable to just keep the syntax simple, refraining from using internalist types in mathematical writing.

Recently, in the setting of dependently typed programming, the distinction between externalism and internalism has been blurred [9]. Using datatype-generic techniques [1], an internalist datatype can be expressed as an ornamentation of a basic datatype, and every ornament induces a predicate on that basic datatype. It then follows that the internalist datatype is isomorphic to the basic datatype restricted by the induced predicate. For now, only those predicates defined inductively on the basic datatype can be derived in this way, but this has already captured the most commonly seen class of predicates. For example, the predicate induced by the ornamentation from natural numbers to finite numbers is the greater-than relation, and we have the isomorphism

$$\mathsf{Fin}\, m \;\cong\; \Sigma\, \mathbb{N}\, (\lambda n \mapsto m > n)\ .$$

Exploiting this isomorphism, dependently typed programmers are granted the freedom to switch between externalism and internalism, whichever suits their purpose best. One particular application is bringing externalist compositionality into internalist datatypes: For internalists, the same data structure with different constraints would be defined as different and formally unrelated datatypes, for each of which the common operations need to be reimplemented in current practice, resulting in dreadful reusability. However, thanks to the isomorphism, programs written for a basic datatype can be upgraded to work with fancier datatypes incorporating multiple constraints in a modular fashion by switching to externalist datatypes for modular composition and then switching back to internalist datatypes. Interestingly, it is argued that the use of externalist predicates can actually be regarded as a standard *internalist* technique — exposing information in the indices — to solve the longstanding reusability problem for internalist programs. Internalism may not be a completely new idea after all.

Type theory makes it possible to mix programs with logic, and thus allows us to bind and manipulate data and proofs together, which is quite different from the programming styles we are used to. We do not know how much potential internalism has, but as its possibility remains to be explored, it seems premature to stick to the traditional, externalist approach that strictly separates programming from logic. Another supporting example is that optimisation of dependently typed programs may exploit value dependencies in types and eliminate a substantial portion of code [4] — the length of a vector does not need to be actually stored, *vhead* can just assume that the input vector starts with a cons node, etc. Program optimisation thus does not necessarily take the form of program extraction, which is based on the distinction between programs and proofs. As McBride said, "[t]here is a tendency to see programming as a fixed notion, essentially untyped. In this view, we make sense of and organise programs by assigning types to them, the way a biologist classifies species, and in order to classify more the exotic creatures, like *printf* or the *zipWith* family,

one requires more exotic types. This conception fails to engage with the full potential of types to make a positive contribution to program construction" [15]. To support this belief, he presented a development of the first-order unification algorithm, which has long been described using general recursion and required separate termination and correctness proofs, as a structurally recursive, dependently typed program which is correct by construction [14]. The moral is that, to truly adopt internalism, we need to reconsider how we design datatypes and write programs, so their correctness are simply manifested in their construction, reducing the need of external justification which can be more awkward to produce.

4 Intensional vs. extensional equality

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

The fundamental equality is formulated as *judgemental equality* in type theory. For intuitionistic mathematics it is the intensional, syntactic equality, also known as *definitional equality*, whereas for classical mathematics it is extensional equality. A characteristic feature of judgemental equality is that it is fully substitutive: Judgementally equal terms can be freely substituted for one another. So after we prove that two sets are extensionally equal in classical mathematics, we can simply substitute one for the other because they are judgementally equal

in the classical, extensional setting. Judgemental equality cannot be expressed as propositions or have proofs inside the theory, though, since it is a meta-theoretical concept, which, for example, is used in type checking in a language implementation and hence is not an entity in the language. To state equality between two objects as a proposition and have proof for that proposition inside the theory, we need *propositional equality*, which can be defined by the following inductive family.

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

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

$$\begin{array}{l} id_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}\\ id_{\mathbb{N}}\ \mathsf{zero}\ \ \ \ = \mathsf{zero}\\ id_{\mathbb{N}}\ (\mathsf{suc}\, n) = \mathsf{suc}\, (id_{\mathbb{N}}\, n)\end{array}$$

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

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

by proving the proposition

$$(n : \mathbb{N}) \to id\, n \equiv id_{\mathbb{N}}\, n\ ,$$

whose proof is by induction on n and thus requires computation. It might be said that propositional equality is "delayed" judgemental equality in propositional form: The terms $id\, n$ (which is definitionally just n) and $id_{\mathbb{N}}\, n$ are not judgementally equal, but they will compute to the same canonical term (and hence become judgementally equal) after substituting a concrete natural number for n, allowing the computation to complete. Streicher suggested [21, p 19] that we "consider the identity type $[t \equiv s]$ as a proposition stating a relation between the *objects denoted by the terms* t and s, respectively, whereas the judgement $t = s \in A$ is a statement of a relation between the *terms* t and s[.]" Indeed, in an intensional setting, if we regard canonical terms to be the semantic objects denoted by terms, then it might be said that two terms are judgementally equal if their normal forms are syntactically identical, while two terms are propositionally equal if they can be proved to compute to the same canonical term after instantiating the context to canonical terms, i.e., they denote the same semantic object. Practically, when used for substitution, a proof of an equality proposition needs to be eliminated by applying the following standard eliminator commonly called J.

$$\begin{array}{l} J\ :\ \{A : \mathsf{Set}\}\ \{x : A\}\ (P : (y : A) \to x \equiv y \to \mathsf{Set}) \to\\ \quad P\, x\, \mathsf{refl} \to \forall\ \{y\} \to (eq : x \equiv y) \to P\, y\, eq\end{array}$$

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

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

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

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

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

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

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

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

$$\begin{aligned}
& id \doteq id_{\mathbb{N}} : (n : \mathbb{N}) \to id\ n \equiv id_{\mathbb{N}}\ n \\
& id \doteq id_{\mathbb{N}}\ \mathsf{zero} \quad = \mathsf{refl} \\
& id \doteq id_{\mathbb{N}}\ (\mathsf{suc}\ n) = \{\, \mathsf{suc}\ n \equiv \mathsf{suc}\ (id_{\mathbb{N}}\ n) \,\}
\end{aligned}$$

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

$$cong\ \mathsf{suc}\ (id \doteq id_{\mathbb{N}}\ n)\ \ :\ \ \mathsf{suc}\ n \equiv \mathsf{suc}\ (id_{\mathbb{N}}\ n)$$

where

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

That is, we need to indicate explicitly that we are using an inductively computed result $id \doteq id_{\mathbb{N}}\ n : n \equiv id_{\mathbb{N}}\ n$, which needs to be further modified by $cong\ \mathsf{suc}$ to match the goal type. On the other hand, if we are working extensionally, a simple refl suffices! The typechecker is told by our placement of refl that $\mathsf{suc}\ n$ and $\mathsf{suc}\ (id_{\mathbb{N}}\ n)$ are actually judgementally equal, and has to somehow figure out that there is a term that has type $n \equiv id_{\mathbb{N}}\ n$, so n and $id_{\mathbb{N}}\ n$ are judgementally equal by equality reflection, and thus $\mathsf{suc}\ n$ and $\mathsf{suc}\ (id_{\mathbb{N}}\ n)$ are indeed judgementally equal by congruence. This example illustrates that type checking in an extensional setting cannot simply resort to syntactic equality of normal forms but needs to search for arbitrary equality proofs. The typechecker can ask for hints from the programmer, like in Sheard's Ωmega language [20, Section 6], but type checking becomes undecidable in general. Another perspective to look at this problem is that the rewriting system underlying judgemental equality loses confluence, since different normal forms may be equated due to the equality reflection rule. Consequently, checking syntactic equality of normal forms is no longer a sound way to do type checking. Losing confluence also means that the computational meaning is disrupted since term reduction becomes nondeterministic. Therefore, the reasoning power offered by extensional type theory may be tempting, but to preserve good computational behaviour we have to stick to intensional type theory, namely giving up the equality reflection rule and keeping judgemental equality intensional.

Notice that even though we have to stick to intensional type theory, we can still reason about extensional properties, like in intuitionistic mathematics. It is just inconvenient to do so. For example, we can always define an extensional equivalence relation stating what it means for two functions to be extensionally equal and use that relation throughout the proofs, but we are not entitled to do substitution, since substitution does not necessarily respect any relation we define. Losing the ability to do substitution, extensional reasoning soon become too heavyweight and difficult to use. A classic example is W-types (for *well orderings*) [12], which can be used to capture all inductive types naturally in extensional type theory but much less conveniently in intensional type theory.

$$\begin{array}{l}\mathbf{data}\ \mathsf{W}\ (A : \mathsf{Set})\ (B : A \to \mathsf{Set}) : \mathsf{Set}\ \mathbf{where}\\ \quad \_\lhd\_ : (a : A) \to (B\ a \to \mathsf{W}\ A\ B) \to \mathsf{W}\ A\ B\end{array}$$

An element of $\mathsf{W}\ A\ B$ is a possibly infinitely branching tree but whose depth is finite. The type of natural numbers, for example, can be defined by

$$Nat = \mathsf{W}\ \mathsf{Bool}\ (\lambda b \mapsto \mathit{if}\ b\ \mathit{then}\ \top\ \mathit{else}\ \bot)$$

where $\top$ is a one-element set whose only constructor is tt and $\bot$ is a set with no elements. Zero may be defined by

$$zero = \mathsf{false} \lhd \lambda() : Nat$$

where $\lambda()$ stands for a function whose domain is empty. The successor function may be defined by

$$suc = \lambda n \mapsto \mathsf{true} \lhd (\lambda\_ \mapsto n) : Nat \to Nat \ .$$

In intensional type theory, however, we run into trouble when we try to define the elimination principle for Nat.

$$\begin{aligned}
&elimNat : (P : Nat \to \mathsf{Set}) \to \\
&\quad P\ zero \to (\forall\ n \to P\ n \to P\ (suc\ n)) \to \forall\ n \to P\ n \\
&elimNat\ \ P\ \ pz\ \ ps\ \ (\mathsf{false} \lhd f) = \{\, P\ (\mathsf{false} \lhd f)\,\}_0 \\
&elimNat\ \ P\ \ pz\ \ ps\ \ (\mathsf{true}\ \lhd f) = \{\, P\ (\mathsf{true}\ \lhd f)\,\}_1
\end{aligned}$$

In goal 0 we wish to fill in $pz : P\ (\mathsf{false} \lhd \lambda())$ to satisfy the goal type $P\ (\mathsf{false} \lhd f)$, but the two types are not the same because f is not necessarily intensionally equal to $\lambda()$. Goal 1 has a similar problem. We get the problem because we claim that $zero$ and $suc\ n$ cover all the elements in Nat, but intensionally that is not the case, since $zero$ and suc are just a specific implementation of the extensional zero and successor function; there are other intensionally different implementations, leading to elements which we cannot generate from $zero$ and suc. We might instead formulate the elimination principle as

$$\begin{aligned}
&elimNat' : (P : Nat \to \mathsf{Set}) \to \\
&\quad (\forall\ f \to P\ (\mathsf{false} \lhd f) \equiv P\ zero) \to (\forall\ f \to P\ (\mathsf{true} \lhd f) \equiv P\ (suc\ (f\ \mathsf{tt}))) \to \\
&\quad P\ zero \to (\forall\ n \to P\ n \to P\ (suc\ n)) \to \forall\ n \to P\ n \\
&elimNat'\ \ P\ \ cz\ \ cs\ \ pz\ \ ps\ \ (\mathsf{false} \lhd f)\ \mathbf{rewrite}\ cz\ f = pz \\
&elimNat'\ \ P\ \ cz\ \ cs\ \ pz\ \ ps\ \ (\mathsf{true}\ \lhd f)\ \mathbf{rewrite}\ cs\ f = \\
&\qquad\qquad ps\ (f\ \mathsf{tt})\ (elimNat'\ P\ cz\ cs\ pz\ ps\ (f\ \mathsf{tt}))\ ,
\end{aligned}$$

requiring that P respects extensional equality,[1] or as

$$\begin{aligned}
&elimNat'' : (P : Nat \to \mathsf{Set}) \to \\
&\quad (\forall\ f \to P\ (\mathsf{false} \lhd f)) \to (\forall\ f \to P\ (f\ \mathsf{tt}) \to P\ (\mathsf{true} \lhd f)) \to \forall\ n \to P\ n \\
&elimNat''\ \ P\ \ pz\ \ ps\ \ (\mathsf{false} \lhd f) = pz\ f \\
&elimNat''\ \ P\ \ pz\ \ ps\ \ (\mathsf{true}\ \lhd f) = ps\ f\ (elimNat''\ P\ pz\ ps\ (f\ \mathsf{tt}))\ ,
\end{aligned}$$

explicitly requiring that P holds for all intensional elements of Nat. But both approaches are rather inconvenient and the premises quickly become tedious to specify when we move on to more complicated types.

Fortunately, having to stick to intensional type theory does not imply that we have to give up the substitutive power of extensional type theory entirely. Altenkirch, McBride, and Swierstra proposed *observational type theory* [2], which integrates extensional propositional equality into intensional type theory and allows extensional substitutions. The new propositional equality is defined to be extensional by induction on the syntax of the types. For example, for functions $f, g : A \to B$ the proposition $f \equiv g$ specialises to $\forall\ x\ y \to x \equiv y \to f\ x \equiv g\ y$. This is just using datatype-generic techniques to generate extensional equality propositions automatically; more interesting modifications to the theory need to

[1] Here propositional equality between *sets* is used, which in fact requires another definition of $\_\equiv\_$ which is a level higher in the universe hierarchy.

be introduced to allow substitution of extensionally equal terms. Instead of the equality reflection rule, which allows silent substitutions, Altenkirch et al. introduced an explicit piece of syntax for coercing values between propositionally equal sets:

$$\frac{x : S \qquad Q : S \equiv T}{x\ [q : S \equiv T\rangle : T}$$

The coercion operator $[q : S \equiv T\rangle$ is equipped with suitable computational rules also defined by induction on the syntax of the types such that it vanishes when all the terms involved are canonical. Now for any predicate P over $A \to B$ and functions $f, g : A \to B$, $P\ f$ and $P\ g$ are not judgementally equal, but we can *postulate* that they are propositionally equal without losing consistency [8]. This means that in general we have another constant:

$$\frac{S : \mathsf{Set} \qquad T : S \to \mathsf{Set}}{[\mathsf{R}x : S.\ T\ x] : \forall\ x\ y \to x \equiv y \to T\ x \equiv T\ y}$$

For example, the two added premises of *elimNat′* regarding extensionality of P can be directly discharged using this constant, so *elimNat′* reduces to *elimNat*, as we originally desired. The substitution operator can then be defined:

$$\begin{aligned} &\mathit{subst} : \{S : \mathsf{Set}\}\ (T : S \to \mathsf{Set}) \to \forall\ \{x\ y\} \to x \equiv y \to T\ x \to T\ y \\ &\mathit{subst}\ \{S\}\ T\ \{x\}\ \{y\}\ \mathit{eq}\ t = t\ [\,[\mathsf{R}z : S.\ T\ z]\ x\ y\ \mathit{eq} : T\ x \equiv T\ y\rangle \end{aligned}$$

How does *subst* compute, however? We abandoned extensional type theory because good computational behaviour was lost, so we had better be sure that it is preserved in observational type theory. The constant R does not have associated computation rules, so the reduction of the whole term can get stuck at the constant and cannot be brought into canonical form if at some point we try to compute the equality proof. Losing canonicity, i.e., not every term reduces to canonical form, again would mean that the computational meaning of the theory is disrupted. But in fact we do not need to inspect the proof! All we need to know is that there *is* a proof that $T\ x \equiv T\ y$, and then the coercion operator necessarily acts like an identity function — the content of the proof cannot affect computation in any way and therefore does not matter. Since the computational behaviour of the coercion operator does not depend on equality proofs, these proofs can be safely erased at runtime, and we can postulate any laws about equality as long as consistency is not broken, which justifies our introduction of the constant R without saying how it computes. The judgemental equality needs to be modified accordingly, but it remains decidable — and so does type checking. We thus obtain an extensional substitution mechanism in an intensional setting. The datatype-generically generated extensional propositional equality is just one of the possible kinds of equality we use in practice; it is hoped that the theory can be extended so programmers can supply their own equivalence relation — forming quotient types — and some guarantees that the relation is respected, and then enjoy the convenience of reasoning offered by the substitution mechanism of observational type theory.

Summary

We have seen that intuitionistic type theory is based on a notion of computation and thereby integrates mathematical objects and logical proofs coherently in one

theory, recognising that they are of the same nature intuitionistically. It serves as a suitable foundation for both intuitionistic mathematics and dependently typed programming, although facilities need to be developed to make a practical programming language: Nontrivial extensions of the theory are needed, e.g., introducing observational equality to make extensional reasoning more convenient; language and tool support are needed, e.g., providing dependent pattern matching and an interactive development environment to help programmers to deal with a more complicated type system; new programming techniques and paradigms are needed, e.g., programming with internalist datatypes so that the correctness of programs are manifested in their construction, reducing the need for separate proofs.

References

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Programming Languages meets Program Verification*, PLPV '07, pages 57–68. ACM, 2007.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[4] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, 2004.

[5] Michael Dummett. *Elements of Intuitionism*. Clarendon Press, 1977.

[6] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer-Verlag, 2006.

[7] Arend Heyting. *Intuitionism: An Introduction*. Amsterdam: North-Holland Publishing, third revised edition, 1971.

[8] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[9] Hsiang-Shang Ko and Jeremy Gibbons. Modularising inductive families. In *Workshop on Generic Programming*, WGP '11, September 2011. To appear.

[10] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press, 1994.

[11] Per Martin-Löf. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London*, 312(1522):501–518, October 1984.

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

[13] Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420, December 1987.

[14] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

[15] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2004.

[16] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.

[17] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[18] Ulf Norell. Dependently typed programming in Agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming (AFP 2008)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

[19] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM, January 1989.

[20] Tim Sheard and Nathan Linger. Programming in Ωmega. In Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, editors, *Second Central-European Functional Programming School (CEFP '07)*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer-Verlag, 2007.

[21] Thomas Streicher. Investigations into intentional type theory. Habilitation thesis, Ludwig Maximilian Universität, November 1993.

[22] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313, 1987.

Appendix C

Modularising inductive families

This paper has been accepted for publication at the ACM Workshop on Generic Programming in Tokyo, 18th September 2011. It was co-authored with Jeremy Gibbons, although the technical contributions are mine.

Modularising Inductive Families

Hsiang-Shang Ko Jeremy Gibbons
Department of Computer Science, University of Oxford
{Hsiang-Shang.Ko, Jeremy.Gibbons}@cs.ox.ac.uk

Abstract

Dependently typed programmers are encouraged to use inductive families to integrate constraints with data construction. Different constraints are used in different contexts, leading to different versions of datatypes for the same data structure. Modular implementation of common operations for these structurally similar datatypes has been a longstanding problem. We propose a datatype-generic solution based on McBride's datatype ornaments [11], exploiting an isomorphism whose interpretation borrows ideas from realisability. Relevant properties of the operations are separately proven for each constraint, and after the programmer selects several constraints to impose on a basic datatype and synthesises an inductive family incorporating those constraints, the operations can be routinely upgraded to work with the synthesised inductive family.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Design, Languages, Theory

Keywords Dependently Typed Programming, Inductive Families, Datatype-Generic Programming

1. Introduction

Dependently typed programmers are encouraged to use *inductive families*, i.e., datatypes with fancy indices, to integrate various constraints with data construction. Correctness proofs are built into and manipulated simultaneously with the data, and in ideal cases correct programs can be written in blissful ignorance of the proofs. We might characterise this approach as *internalist*, suggesting that data constraints are internalised. In contrast, the more traditional approach which favours using only basic datatypes and expressing constraints through separate predicates on those datatypes might be described as *externalist*.

The internalist approach easily leads to an explosion in differently indexed versions of the same data structure. For example, as well as ordinary lists, in different contexts we may need vectors (lists indexed with their length), sorted lists, or sorted vectors, ending up with four slightly different but structurally similar datatypes. The problem, then, is how the common operations are implemented for these different versions of the datatype. Current practice is to completely reimplement the operations for each version, causing serious code duplication and dreadful reusability. The externalist approach, in contrast, responds to this problem very well. We would have only one basic list type, with one predicate stating that a list has a certain length and another predicate asserting that a list is sorted. The list type is upgraded to the vector type, the sorted list type, or the sorted vector type by simply pairing the list type with the sortedness predicate, the length predicate, or the pointwise conjunction of the two predicates. The common operations are implemented for ordinary lists only, and their properties regarding ordering or length are separately proven and invoked when needed. Can we somehow introduce this beneficial compositionality to internalism as well? Yes, we can! There is an isomorphism between externalist and internalist datatypes to be exploited.

To illustrate, let us go through a small case study about upgrading a function on natural numbers. The internalists use the following datatype to characterise the *finite numbers*, which are natural numbers bounded above by a certain number.

$$\begin{array}{l}\textbf{data}\ \mathsf{Fin} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{fzero} : \{m : \mathsf{Nat}\} \to \mathsf{Fin}\ (\mathsf{suc}\ m)\\ \quad \mathsf{fsuc} \ : \{m : \mathsf{Nat}\} \to \mathsf{Fin}\ m \to \mathsf{Fin}\ (\mathsf{suc}\ m)\end{array}$$

We can be explicit about how we regard finite numbers as natural numbers by defining a forgetful map.

$$\begin{array}{l}\mathit{forget}_{\mathsf{F}} : \forall\ \{m\} \to \mathsf{Fin}\ m \to \mathsf{Nat}\\ \mathit{forget}_{\mathsf{F}}\ \mathsf{fzero}\ \ \ = \mathsf{zero}\\ \mathit{forget}_{\mathsf{F}}\ (\mathsf{fsuc}\ i) = \mathsf{suc}\ (\mathit{forget}_{\mathsf{F}}\ i)\end{array}$$

To represent the same type, externalists would first define a greater-than relation for natural numbers,

$$\begin{array}{l}\textbf{data}\ \_>\_ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{base} : \{m : \mathsf{Nat}\} \to \mathsf{suc}\ m > \mathsf{zero}\\ \quad \mathsf{step} : \{m\ n : \mathsf{Nat}\} \to m > n \to \mathsf{suc}\ m > \mathsf{suc}\ n\ ,\end{array}$$

and then use the dependent pair type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto m > n)$, an object of which is a natural number n paired with a proof that $m > n$. We have an isomorphism between the two types,

$$\mathsf{Fin}\ m\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto m > n)\ ,$$

witnessed by

$$\begin{array}{l}\mathfrak{R}_{\mathsf{F}} : \forall\ \{m\} \to (i : \mathsf{Fin}\ m) \to m > \mathit{forget}_{\mathsf{F}}\ i\\ \mathfrak{R}_{\mathsf{F}}\ \mathsf{fzero}\ \ \ = \mathsf{base}\\ \mathfrak{R}_{\mathsf{F}}\ (\mathsf{fsuc}\ i) = \mathsf{step}\ (\mathfrak{R}_{\mathsf{F}}\ i)\end{array}$$

and

$$\begin{array}{l}\mathfrak{R}_{\mathsf{F}}^{-1} : \forall\ \{m\} \to (n : \mathsf{Nat}) \to m > n \to \mathsf{Fin}\ m\\ \mathfrak{R}_{\mathsf{F}}^{-1}\ .\mathsf{zero}\ \ \ \mathsf{base}\ \ \ \ \ = \mathsf{fzero}\\ \mathfrak{R}_{\mathsf{F}}^{-1}\ .(\mathsf{suc}\ \_)\ (\mathsf{step}\ gt) = \mathsf{fsuc}\ (\mathfrak{R}_{\mathsf{F}}^{-1}\ \_\ gt)\ .\end{array}$$

Now suppose that we have some function $f' : \mathsf{Nat} \to \mathsf{Nat}$, and additionally that we can prove externally that f' preserves upper bounds (in other words, is non-increasing):

$$f'\text{-}\mathit{bound} : \forall\ \{m\ n\} \to m > n \to m > f'\ n\ .$$

WGP'11, September 18, 2011, Tokyo, Japan.

Then we can upgrade f' to work with finite numbers by exploiting the isomorphism:

$$\begin{array}{l} f_{\mathsf{F}} : \forall\ \{m\} \to \mathsf{Fin}\ m \to \mathsf{Fin}\ m \\ f_{\mathsf{F}}\ i\ =\ \mathfrak{R}_{\mathsf{F}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{F}}\ i))\ (f'\text{-}\mathit{bound}\ (\mathfrak{R}_{\mathsf{F}}\ i))\ . \end{array}$$

The input finite number $i : \mathsf{Fin}\ m$ is split into the underlying natural number $\mathit{forget}_{\mathsf{F}}\ i : \mathsf{Nat}$ and a corresponding proof $\mathfrak{R}_{\mathsf{F}}\ i : m > \mathit{forget}\ i$. The natural number is then processed by f' and the proof by f'-*bound*, before the results are integrated back into a finite number by way of $\mathfrak{R}_{\mathsf{F}}^{-1}$.

Further suppose that we need parity information about f'. The externalists would define a function to compute the parity of a natural number,

$$\begin{array}{l} \mathit{parity} : \mathsf{Nat} \to \mathsf{Bool} \\ \mathit{parity}\ \mathsf{zero}\ \ \ \ =\ \mathsf{false} \\ \mathit{parity}\ (\mathsf{suc}\ n)\ =\ \mathit{not}\ (\mathit{parity}\ n)\ , \end{array}$$

and use the type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto \mathit{parity}\ n \equiv b)$ (where $\_\equiv\_$ is the propositional equality type) for those natural numbers of parity b. The internalists would define a new datatype

$$\begin{array}{l} \textbf{data}\ \mathsf{PNat} : \mathsf{Bool} \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{pzero}\ :\ \mathsf{PNat}\ \mathsf{false} \\ \quad \mathsf{psuc}\ \ :\ \{b : \mathsf{Bool}\} \to \mathsf{PNat}\ b \to \mathsf{PNat}\ (\mathit{not}\ b)\ , \end{array}$$

and use $\mathsf{PNat}\ b$ for the same set of natural numbers. Assume f' preserves parity, i.e., we can prove

$$f'\text{-}\mathit{parity} : \forall\ \{n\ b\} \to \mathit{parity}\ n \equiv b \to \mathit{parity}\ (f'\ n) \equiv b\ .$$

Following the same recipe, by exploiting the isomorphism

$$\mathsf{PNat}\ b\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto \mathit{parity}\ n \equiv b)$$

witnessed by

$$\begin{array}{l} \mathit{forget}_{\mathsf{P}} : \forall\ \{b\} \to \mathsf{PNat}\ b \to \mathsf{Nat} \\ \mathit{forget}_{\mathsf{P}}\ \mathsf{pzero}\ \ \ \ =\ \mathsf{zero} \\ \mathit{forget}_{\mathsf{P}}\ (\mathsf{psuc}\ j)\ =\ \mathsf{suc}\ (\mathit{forget}_{\mathsf{P}}\ j) \end{array}$$

$$\begin{array}{l} \mathfrak{R}_{\mathsf{P}} : \forall\ \{b\} \to (j : \mathsf{PNat}\ b) \to \mathit{parity}\ (\mathit{forget}_{\mathsf{P}}\ j) \equiv b \\ \mathfrak{R}_{\mathsf{P}}\ \mathsf{pzero}\ =\ \mathsf{refl} \\ \mathfrak{R}_{\mathsf{P}}\ (\mathsf{psuc}\ j)\ \textbf{rewrite}\ \mathfrak{R}_{\mathsf{P}}\ j\ =\ \mathsf{refl} \end{array}$$

and

$$\begin{array}{l} \mathfrak{R}_{\mathsf{P}}^{-1} : \forall\ \{b\} \to (n : \mathsf{Nat}) \to \mathit{parity}\ n \equiv b \to \mathsf{PNat}\ b \\ \mathfrak{R}_{\mathsf{P}}^{-1}\ \mathsf{zero}\ \ \ \ \mathsf{refl}\ =\ \mathsf{pzero} \\ \mathfrak{R}_{\mathsf{P}}^{-1}\ (\mathsf{suc}\ n)\ \mathsf{refl}\ =\ \mathsf{psuc}\ (\mathfrak{R}_{\mathsf{P}}^{-1}\ n\ \mathsf{refl})\ , \end{array}$$

we can again upgrade f' to work with PNat:

$$\begin{array}{l} f_{\mathsf{P}} : \forall\ \{b\} \to \mathsf{PNat}\ b \to \mathsf{PNat}\ b \\ f_{\mathsf{P}}\ j\ =\ \mathfrak{R}_{\mathsf{P}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{P}}\ j))\ (f'\text{-}\mathit{parity}\ (\mathfrak{R}_{\mathsf{P}}\ j))\ . \end{array}$$

Finally, consider finite numbers with parity information. The externalists would simply put the two predicates together and get the type $\Sigma\ \mathsf{Nat}\ (\lambda n \mapsto (m > n) \times (\mathit{parity}\ n \equiv b))$ for the natural numbers bounded above by m and of parity b. The internalists would define yet another datatype

$$\begin{array}{l} \textbf{data}\ \mathsf{PFin} : \mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{pfzero}\ :\ \forall\ \{m\} \to \mathsf{PFin}\ (\mathsf{suc}\ m)\ \mathsf{false} \\ \quad \mathsf{pfsuc}\ \ :\ \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{PFin}\ (\mathsf{suc}\ m)\ (\mathit{not}\ b) \end{array}$$

and use $\mathsf{PFin}\ m\ b$ for the same set of natural numbers. We still have an isomorphism

$$\mathsf{PFin}\ m\ b\ \cong\ \Sigma\ \mathsf{Nat}\ (\lambda n \mapsto (m > n) \times (\mathit{parity}\ n \equiv b))$$

witnessed by

$$\begin{array}{l} \mathit{forget}_{\mathsf{PF}} : \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{Nat} \\ \mathit{forget}_{\mathsf{PF}}\ \mathsf{pfzero}\ \ \ \ =\ \mathsf{zero} \\ \mathit{forget}_{\mathsf{PF}}\ (\mathsf{pfsuc}\ k)\ =\ \mathsf{suc}\ (\mathit{forget}_{\mathsf{PF}}\ k) \end{array}$$

$$\begin{array}{l} \mathfrak{R}_{\mathsf{PF}}\text{-}l : \forall\ \{m\ b\} \to (k : \mathsf{PFin}\ m\ b) \to m > \mathit{forget}_{\mathsf{PF}}\ k \\ \mathfrak{R}_{\mathsf{PF}}\text{-}l\ \mathsf{pfzero}\ \ \ \ =\ \mathsf{base} \\ \mathfrak{R}_{\mathsf{PF}}\text{-}l\ (\mathsf{pfsuc}\ k)\ =\ \mathsf{step}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}l\ k) \end{array}$$

$$\begin{array}{l} \mathfrak{R}_{\mathsf{PF}}\text{-}r : \forall\ \{m\ b\} \to (k : \mathsf{PFin}\ m\ b) \to \mathit{parity}\ (\mathit{forget}_{\mathsf{PF}}\ k) \equiv b \\ \mathfrak{R}_{\mathsf{PF}}\text{-}r\ \mathsf{pfzero}\ =\ \mathsf{refl} \\ \mathfrak{R}_{\mathsf{PF}}\text{-}r\ (\mathsf{pfsuc}\ k)\ \textbf{rewrite}\ \mathfrak{R}_{\mathsf{PF}}\text{-}r\ k\ =\ \mathsf{refl} \end{array}$$

and

$$\begin{array}{l} \mathfrak{R}_{\mathsf{PF}}^{-1} : \forall\ \{m\ b\} \to (n : \mathsf{Nat}) \to m > n \to \mathit{parity}\ n \equiv b \to \mathsf{PFin}\ m\ b \\ \mathfrak{R}_{\mathsf{PF}}^{-1}\ .\mathsf{zero}\ \ \ \ \mathsf{base}\ \ \ \ \ \ \mathsf{refl}\ =\ \mathsf{pfzero} \\ \mathfrak{R}_{\mathsf{PF}}^{-1}\ .(\mathsf{suc}\ \_)\ (\mathsf{step}\ \mathit{gt})\ \mathsf{refl}\ =\ \mathsf{pfsuc}\ (\mathfrak{R}_{\mathsf{PF}}^{-1}\ \_\ \mathit{gt}\ \mathsf{refl})\ , \end{array}$$

and the isomorphism can again be used to upgrade f' to work with PFin, but this time the proof part *reuses* the existing proofs f'-*bound* and f'-*parity*:

$$\begin{array}{l} f_{\mathsf{PF}} : \forall\ \{m\ b\} \to \mathsf{PFin}\ m\ b \to \mathsf{PFin}\ m\ b \\ f_{\mathsf{PF}}\ k\ =\ \mathfrak{R}_{\mathsf{PF}}^{-1}\ (f'\ (\mathit{forget}_{\mathsf{PF}}\ k)) \\ \qquad\qquad\qquad (f'\text{-}\mathit{bound}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}l\ k))\ (f'\text{-}\mathit{parity}\ (\mathfrak{R}_{\mathsf{PF}}\text{-}r\ k))\ . \end{array}$$

Had we implemented f_{F} and f_{P} directly instead of exploiting the isomorphisms, it would have been much less straightforward to synthesise f_{PF} from them. It is thanks to the isomorphism maps $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ that we can routinely synthesise f_{F} and f_{P} from corresponding externalist proofs, and — more interestingly — that we can develop f_{PF} modularly, reusing those externalist proofs. The reusability problem is thus reduced to writing the isomorphisms, and the good news is that the isomorphisms can be synthesised *datatype-generically*. Acquiring the power of datatype-generic programming, we can even synthesise PFin from the ingredients used to make Fin and PNat out of Nat, revealing the same compositional structure of the internalist types corresponding to that of their externalist brethren.

Outline of this paper. Our work is heavily based on McBride's datatype *ornaments* [11], which provide a datatype-generic language in which to talk about the relationship among structurally similar datatypes. McBride's work is summarised in Section 2. An ornament describes how to upgrade a basic datatype to a fancier one, often embedding some constraints into data construction. Then an interpretation based on realisability is given in Section 3: Given an ornament, objects of the basic datatype are considered as incomplete proofs of the fancier datatype, and the information needed to restore a complete proof from an incomplete one is stated by the *realisability predicate* induced by the ornament. With the interpretation, we are enabled to think about *composition of ornaments*, and thus indexed datatypes with multiple constraints, in terms of pointwise conjunction of realisability predicates. As an initial experiment, in Section 4 we consider the special case where one of the two ornaments being composed is *algebraic*. We prove that the pointwise conjunction of the realisability predicates induced by the component ornaments is isomorphic to the realisability predicate induced by the composite ornament, and demonstrate how this helps to write functions on indexed datatypes incorporating multiple constraints in a modular style. Section 5 discusses how the interpretation connects internalism and externalism, and how we might exploit this connection to structure our libraries. Section 6 compares ours with previous work, and finally Section 7 presents some possible future directions. We have implemented our ideas in Agda [14], source available at `http://www.cs.ox.ac.uk/people/hsiang-shang.ko/OAOAOO/`.

2. A recapitulation of datatype ornaments

To state the realisability interpretation generically, first we need a *datatype-generic* framework for talking about the relationship between structurally similar datatypes. Central to datatype-generic programming is the idea that the structure of datatypes can be coded as first-class entities and thus become ordinary parameters to programs. The same idea is also found in Martin-Löf's Type Theory [10], in which a set of codes for datatypes is called a *universe* (à la Tarski), and there is a decoding function translating codes to actual types. Type theory being the foundation of dependently typed languages, universe construction can be implemented directly in such languages, so datatype-generic programming becomes just ordinary programming in the dependently typed world [1].

McBride's seminal work on datatype ornaments [11] is ideally suited to our purposes. What he did was to construct a universe in Agda, i.e., a datatype whose inhabitants are codes to be translated into actual types, with generic fold and induction for decoded types, and define another datatype whose inhabitants — called *ornaments* — explain how to "patch" a code to a richer one but retaining the basic structure. For example, a list is a Peano-style natural number whose successor nodes are decorated with elements, and a vector is a list whose type is indexed with its length. Ornaments are designed to encode these two kinds of addition of information: *decoration* (element insertion) and *refinement* (index upgrade). Consequently, induced by every ornament is a forgetful map erasing the added information from an object of the ornamented datatype and recovering an object of the raw datatype. For example, the forgetful map induced by the ornamentation from natural numbers to lists is just *length*, which discards the elements associated with the cons nodes. The forgetful map is a fold, and the algebra of the fold is called the *ornamental algebra*, as it is induced by an ornament. Conversely, every algebra induces an *algebraic ornament*, which provides a systematic way to index the type of an object with the result of the fold of the algebra applied to that object. The vector type is a typical example — it arises from the algebraic ornamentation of lists which indexes the type of a list with its length.

Datatype descriptions. Concretely, McBride used the datatype

$$\begin{array}{l}\textbf{data}\ \mathsf{Desc}\ (I : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where}\\ \quad \mathsf{say} \quad\ : I \to \mathsf{Desc}\ I\\ \quad \sigma \quad\ : (S : \mathsf{Set}) \to (S \to \mathsf{Desc}\ I) \to \mathsf{Desc}\ I\\ \quad \mathsf{ask\_*\_} : I \to \mathsf{Desc}\ I \to \mathsf{Desc}\ I\end{array}$$

as the universe. A term of type $\mathsf{Desc}\ I$ describes an inductive family of type $I \to \mathsf{Set}$ by specifying how its data are constructed: The first constructor $\mathsf{say}\ i$ marks the end of a description and delivers data at index i; the second constructor $\sigma\ S\ D$ inserts an element of type S on which the remaining description D may depend; the third constructor $\mathsf{ask}\ i * D$ recursively requests data at index i and then continues with D. For example, the code for the type of natural numbers is

$$\begin{array}{l}\mathit{NatD}\ :\ \mathsf{Desc}\ \top\\ \mathit{NatD}\ =\ \sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ \mathsf{tt}\\ \qquad\qquad\qquad\qquad \mathit{true} \mapsto\ \mathsf{ask}\ \mathsf{tt} * \mathsf{say}\ \mathsf{tt})\ ,\end{array}$$

where $\top$ is a one-element type whose only constructor is tt, and $\mathit{false} \mapsto \_\ \mathit{true} \mapsto \_$ is a function imitating dependent case expressions,

$$\begin{array}{l}\mathit{false} \mapsto \_\ \mathit{true} \mapsto \_ : \{P : \mathsf{Bool} \to \mathsf{Set}_1\} \to\\ \qquad\qquad P\ \mathsf{false} \to P\ \mathsf{true} \to (b : \mathsf{Bool}) \to P\ b\\ (\mathit{false} \mapsto p\ \mathit{true} \mapsto q)\ \mathsf{false}\ =\ p\\ (\mathit{false} \mapsto p\ \mathit{true} \mapsto q)\ \mathsf{true}\ =\ q\ .\end{array}$$

The description *NatD* describes exactly how to construct a Peano-style natural number: We choose one constructor out of two by giving a boolean value; if it is false, the construction is complete and the result is delivered at the trivial index tt; otherwise it is true, in which case we recursively ask for a natural number before delivering the result.

To translate a description of type $\mathsf{Desc}\ I$ to an actual type, first we decode it to an endofunctor on $I \to \mathsf{Set}$.

$$\begin{array}{l}[\![\_]\!] : \{I : \mathsf{Set}\} \to \mathsf{Desc}\ I \to (I \to \mathsf{Set}) \to I \to \mathsf{Set}\\ [\![\mathsf{say}\ i]\!] \qquad X\ i' = i \equiv i'\\ [\![\sigma\ S\ D]\!] \quad X\ i' = \Sigma\ S\ \lambda s \mapsto [\![D\ s]\!]\ i'\\ [\![\mathsf{ask}\ i * D]\!]\ X\ i' = X\ i \times [\![D]\!]\ i'\end{array}$$

Then we can take the least fixed point of the decoded functor by the following native inductive datatype:

$$\begin{array}{l}\textbf{data}\ \mu\ \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) : I \to \mathsf{Set}\ \textbf{where}\\ \quad \langle\_\rangle\ :\ \forall\ \{i\} \to [\![D]\!]\ (\mu\ D)\ i \to \mu\ D\ i\ .\end{array}$$

If we introduce a notation for functions on $I \to \mathsf{Set}$,

$$\begin{array}{l}\_\Rightarrow\_ : \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ X \Rightarrow Y\ =\ \forall\ \{i\} \to X\ i \to Y\ i\ ,\end{array}$$

we see that $\langle\_\rangle : [\![D]\!]\ (\mu\ D) \Rightarrow \mu\ D$ has the familiar form of an algebra for the functor $[\![D]\!]$, which is in fact the initial algebra. So the type of natural numbers, *Nat*, is obtained by decoding *NatD*.[1]

$$\begin{array}{l}\mathit{Nat}\ :\ \mathsf{Set}\\ \mathit{Nat}\ =\ \mu\ \mathit{NatD}\ \mathsf{tt}\end{array}$$

The decoded type *Nat* being a native inductive type, we can define functions on such natural numbers by pattern matching, albeit a bit cryptically, like

$$\begin{array}{l}\mathit{pred} : \mathit{Nat} \to \mathit{Nat}\\ \mathit{pred}\ \langle \mathsf{false}, \mathsf{refl} \rangle\ =\ \mathit{zero}\\ \mathit{pred}\ \langle \mathsf{true}, n, \mathsf{refl} \rangle\ =\ n\end{array}$$

where $\mathit{zero} = \langle \mathsf{false}, \mathsf{refl} \rangle : \mathit{Nat}$. But later when we need to define operations and state properties for all the types encoded by the universe, it is necessary to have a generic fold operator parametrised by the codes:

$$\mathit{fold} : \{I\ X : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\} \to ([\![D]\!]\ X \Rightarrow X) \to \mu\ D \Rightarrow X\ .$$

There is also a generic induction operator, which is more powerful and subsumes generic fold, but fold is much easier to use when the full power of induction is not required. The implementation details of the two operators are not essential to our exposition and hence are omitted from this paper.

Ornaments. Next we define the ornaments. An ornament is a "relative" description which is written with respect to another description and marks changes relative to the latter. One of the two kinds of information expressed in ornaments is *refinement*: how to promote the I-indices in an I-description to J-indices with respect to an index erasure function $e : J \to I$ — the new J-indices appearing in an ornament must be erasable by e to the original I-indices. The following inverse-image datatype helps to enforce this requirement:

$$\begin{array}{l}\textbf{data}\ \_^{-1}\_\ \{I\ J : \mathsf{Set}\}\ (e : J \to I) : I \to \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{ok}\ :\ (j : J) \to e^{-1}\ (e\ j)\ .\end{array}$$

If we have a value of type $e^{-1}\ i$, then we are guaranteed to be able to extract from it a value j such that $e\ j$ is definitionally equal to i. The ornaments are then defined as a datatype indexed by descriptions of type $\mathsf{Desc}\ I$. Its first three constructors mirror those of $\mathsf{Desc}\ I$, refining I-indices to J-indices, while the fourth constructor Δ provides the second kind of ornamental information

[1] A typographical convention: Type and data constructors introduced by native data declarations are typeset in sans serif, while other terms like functions, variables, etc. are typeset in *italics*. So the Nat we saw in Section 1 is a native datatype, whereas *Nat* here is a decoded datatype.

on *decoration*, signalling insertion of a new element on which the trailing ornament may depend.

$$\begin{array}{l}\textbf{data}\ \mathsf{Orn}\ \{I : \mathsf{Set}\}\ (J : \mathsf{Set})\ (e : J \to I) : \mathsf{Desc}\ I \to \mathsf{Set}_1\ \textbf{where}\\ \quad \mathsf{say} \quad : \{i : I\} \to e^{-1}\ i \to \mathsf{Orn}\ J\ e\ (\mathsf{say}\ i)\\ \quad \sigma \quad : (S : \mathsf{Set})\ \{D : S \to \mathsf{Desc}\ I\} \to\\ \qquad\qquad (\forall\ s \to \mathsf{Orn}\ J\ e\ (D\ s)) \to \mathsf{Orn}\ J\ e\ (\sigma\ S\ D)\\ \quad \mathsf{ask\_*\_} : \{i : I\} \to e^{-1}\ i \to\\ \qquad\qquad \forall\ \{D\} \to \mathsf{Orn}\ J\ e\ D \to \mathsf{Orn}\ J\ e\ (\mathsf{ask}\ i * D)\\ \quad \Delta \quad : (S : \mathsf{Set})\ \{D : \mathsf{Desc}\ I\} \to\\ \qquad\qquad (S \to \mathsf{Orn}\ J\ e\ D) \to \mathsf{Orn}\ J\ e\ D\end{array}$$

For example, the ornament

$$\begin{array}{l}\mathit{ListO} : \mathsf{Set} \to \mathsf{Orn}\ \top\ \mathit{id}\ \mathit{NatD}\\ \mathit{ListO}\ A =\\ \quad \sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ (\mathsf{ok}\ \mathsf{tt})\\ \qquad\qquad \mathit{true} \mapsto\ \Delta\ A\ \lambda\_ \mapsto \mathsf{ask}\ (\mathsf{ok}\ \mathsf{tt}) * \mathsf{say}\ (\mathsf{ok}\ \mathsf{tt}))\end{array}$$

describes the ornamentation from natural numbers to lists. It looks very much like a description except that the indices are wrapped in ok and the Δ should have been σ. We get these differences because *ListO A* is a description *relative to NatD*: The new indices have to prove that they respect *id* by wrapping themselves in ok and Δ is used in place of σ to indicate that the element is not originally in *NatD*. Generically, an ornament of type $\mathsf{Orn}\ J\ e\ D$ can of course be decoded into an "absolute" description of type $\mathsf{Desc}\ J$ by unwrapping the J-indices and translating Δ to σ:

$$\begin{array}{l}\lfloor\_\rfloor : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\} \to \mathsf{Orn}\ J\ e\ D \to \mathsf{Desc}\ J\\ \lfloor \mathsf{say}\ (\mathsf{ok}\ j)\rfloor \quad = \mathsf{say}\ j\\ \lfloor \sigma\ S\ O\rfloor \quad = \sigma\ S\ \lambda s \mapsto \lfloor O\ s\rfloor\\ \lfloor \mathsf{ask}\ (\mathsf{ok}\ j) * O\rfloor \ = \mathsf{ask}\ j * \lfloor O\rfloor\\ \lfloor \Delta\ S\ O\rfloor \quad = \sigma\ S\ \lambda s \mapsto \lfloor O\ s\rfloor\ .\end{array}$$

So the decoded description $\lfloor \mathit{ListO}\ A\rfloor$ expands to

$$\begin{array}{l}\sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ \mathsf{tt}\\ \qquad\qquad \mathit{true} \mapsto\ \sigma\ A\ \lambda\_ \mapsto \mathsf{ask}\ \mathsf{tt} * \mathsf{say}\ \mathsf{tt})\end{array}$$

as expected, which can then be decoded to the list type $\mathit{List}\ A = \mu\ \lfloor \mathit{ListO}\ A\rfloor\ \mathsf{tt}$.

An ornament $O : \mathsf{Orn}\ J\ e\ D$ gives rise to an *ornamental algebra* $\mathit{ornAlg}\ O : [\![\lfloor O\rfloor]\!]\ (\mu\ D \cdot e) \Rightarrow (\mu\ D \cdot e)$ which erases elements added by Δ and demotes the indices. (The $\_\cdot\_$ operator is function composition.) First we define a polymorphic restructuring map erasing information added by Δ,

$$\begin{array}{l}\mathit{erase} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D)\ \{X\} \to\\ \qquad [\![\lfloor O\rfloor]\!]\ (X \cdot e) \Rightarrow [\![D]\!]\ X \cdot e\\ \mathit{erase}\ (\mathsf{say}\ (\mathsf{ok}\ j))\quad \mathsf{refl} \quad = \mathsf{refl}\\ \mathit{erase}\ (\sigma\ S\ O)\quad (s, ds) \ = s, \mathit{erase}\ (O\ s)\ ds\\ \mathit{erase}\ (\mathsf{ask}\ (\mathsf{ok}\ j) * O)\ (d, ds) \ = d, \mathit{erase}\ O\ ds\\ \mathit{erase}\ (\Delta\ S\ O)\quad (s, ds) \ = \mathit{erase}\ (O\ s)\ ds\ ,\end{array}$$

and then the ornamental algebra is defined by

$$\begin{array}{l}\mathit{ornAlg} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to\\ \qquad [\![\lfloor O\rfloor]\!]\ (\mu\ D \cdot e) \Rightarrow (\mu\ D \cdot e)\\ \mathit{ornAlg}\ \ O\ ds\ = \langle \mathit{erase}\ O\ ds\rangle\ .\end{array}$$

Folding with the ornamental algebra gives us the *forgetful map*

$$\begin{array}{l}\mathit{forget} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to\\ \qquad \mu\ \lfloor O\rfloor \Rightarrow (\mu\ D \cdot e)\\ \mathit{forget}\ \ O\ = \mathit{fold}\ (\mathit{ornAlg}\ O)\ .\end{array}$$

For example, the length of a list is computed by

$$\begin{array}{l}\mathit{length} : \forall\ \{A\} \to \mathit{List}\ A \to \mathit{Nat}\\ \mathit{length}\ \ \{A\}\ = \mathit{forget}\ (\mathit{ListO}\ A)\ .\end{array}$$

Algebraic ornaments. Being first-class data, ornaments can be generated systematically. McBride proposed a class of ornaments induced by algebras: Given $D : \mathsf{Desc}\ I$ and an algebra $\phi : [\![D]\!]\ J \Rightarrow J$, the *algebraic ornament* induced by ϕ is defined by

$$\begin{array}{l}\mathit{algOrn} : \{I : \mathsf{Set}\}\ \{J : I \to \mathsf{Set}\} \to\\ \qquad (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \to \mathsf{Orn}\ (\Sigma\ I\ J)\ \mathsf{proj}_1\ D\\ \mathit{algOrn}\ (\mathsf{say}\ i)\ \phi = \mathsf{say}\ (\mathsf{ok}\ (i, \phi\ \mathsf{refl}))\\ \mathit{algOrn}\ (\sigma\ S\ D)\ \phi = \sigma\ S\ \lambda s \mapsto \mathit{algOrn}\ (D\ s)\ (\Lambda\ \phi\ s)\\ \mathit{algOrn}\ \{J = J\}\ (\mathsf{ask}\ i * D)\ \phi =\\ \qquad \Delta\ (J\ i)\ \lambda\ j \mapsto \mathsf{ask}\ (\mathsf{ok}\ (i, j)) * \mathit{algOrn}\ D\ (\Lambda\ \phi\ \ j)\ ,\end{array}$$

where Λ is the currying operator. It is perhaps easier to understand algebraic ornaments in a specialised scenario. Suppose we are given $f : A \to B \to B$ and $e : B$, which constitute an algebra for folding a list of type $\mathsf{List}\ A$. The algebraic ornamentation of $\mathsf{List}\ A$ induced by that algebra would lead to the following datatype, where the new indices and elements are framed.

$$\begin{array}{l}\textbf{data}\ \mathsf{AlgList} : \boxed{B} \to \mathsf{Set}\ \textbf{where}\\ \quad [] \quad : \mathsf{AlgList}\ \boxed{e}\\ \quad \_::\_ \ : (x : A)\ \boxed{\{b : B\}}\ (xs : \mathsf{AlgList}\ \boxed{b}) \to \mathsf{AlgList}\ \boxed{(f\ x\ b)}\end{array}$$

If we temporarily ignore the framed parts, we see that an AlgList is basically still a list. The difference is that the index of an AlgList is guaranteed to be the result of folding the underlying list using the given algebra: The new index for the type of $[]$ is e, which is the result of folding $[]$; for $\_::\_$, a new element $b : B$ is inserted before the recursive node xs for storing the index which has been inductively computed for xs and can be assumed to be the result of folding xs, so the final index $f\ x\ b$ is the result of folding $x :: xs$. In the generic implementation of *algOrn*, the tuple to be fed to the algebra ϕ is revealed one component at a time in each step of the case analysis, so ϕ acts as an accumulating parameter, accepting the component revealed in each step with the help of Λ, and emitting the final result when the say case is reached and the final component of the tuple, refl, is fed to it. Additionally, in the ask case where we encounter a recursive node, a new element is inserted by Δ for storing the index j that has been inductively computed for that node.

An example is vectors, which are lists indexed by the result of *length*, which is a fold whose algebra is $\mathit{ornAlg}\ (\mathit{ListO}\ A)$, so the ornamentation from lists to vectors is algebraic:

$$\begin{array}{l}\mathit{VecO} : (A : \mathsf{Set}) \to \mathsf{Orn}\ (\top \times \mathit{Nat})\ \mathsf{proj}_1\ \lfloor \mathit{ListO}\ A\rfloor\\ \mathit{VecO}\ \ A\ = \mathit{algOrn}\ \lfloor \mathit{ListO}\ A\rfloor\ (\mathit{ornAlg}\ (\mathit{ListO}\ A))\ .\end{array}$$

It expands to

$$\begin{array}{l}\sigma\ \mathsf{Bool}\ (\mathit{false} \mapsto \mathsf{say}\ (\mathsf{ok}\ (\mathsf{tt}, \mathit{zero}))\\ \qquad\qquad \mathit{true} \mapsto\ \sigma\ A\ \lambda\_ \mapsto \Delta\ \mathit{Nat}\ \lambda n \mapsto\\ \qquad\qquad\qquad \mathsf{ask}\ (\mathsf{ok}\ (\mathsf{tt}, n)) * \mathsf{say}\ (\mathsf{ok}\ (\mathsf{tt}, \mathit{suc}\ n)))\end{array}$$

where $\mathit{suc} = \lambda n \mapsto \langle \mathsf{true}, n, \mathsf{refl}\rangle : \mathit{Nat} \to \mathit{Nat}$. The decoded type $\mathit{Vec}\ A\ n = \mu\ \lfloor \mathit{VecO}\ A\rfloor\ (\mathsf{tt}, n)$ is essentially the same datatype delivered by the following native data declaration:[2]

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

An algebraically ornamented datatype does not carry more information than the raw datatype, but simply exposes some known knowledge in the index, namely the value obtained by folding the

[2] Frequently we translate decoded datatypes into native data declarations in this paper, but it is only for the purpose of exposition — the decoded datatypes have no formal connection with the natively declared datatypes in Agda (as suggested by the use of different fonts). It is hoped that in future dependently typed languages, native data declarations will become syntactic sugar for codes for datatypes, so the distinction between native datatypes and decoded datatypes will disappear [6].

underlying data. Hence there is not only a forgetful map from the ornamented datatype to the raw datatype, as induced by any ornament, but also a *remembering map* converting the raw datatype to the ornamented datatype, computing the index on the fly. The two maps are inverse to each other, meaning that the algebraically ornamented datatype and the raw datatype really are isomorphic. The remembering map can be defined generically,

$$\begin{array}{l} \mathit{remember} : \\ \quad \forall\, \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ \quad \{i\}\ (x' : \mu\ D\ i) \to \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, \mathit{fold}\ \phi\ x')\ , \end{array}$$

whose implementation is by generic induction and is omitted here.

The type of *remember* states what the index would be when raw data are converted to algebraically ornamented data, namely the result of folding the raw data. Conversely, when algebraically ornamented data are converted to raw data, the *recomputation lemma* states that the forgotten index can be recovered by folding the raw data.

$$\begin{array}{l} \mathit{recomputation} : \\ \quad \forall\, \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ \quad \{ij : \Sigma\ I\ J\}\ (x : \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ ij) \to \\ \qquad \mathit{fold}\ \phi\ (\mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ x) \equiv \mathsf{proj}_2\ ij \end{array}$$

The implementation is again by generic induction and is omitted.

Algebraically ornamented datatypes provide an internalist way of constructing an object specified by requiring the result of folding that object to be a predetermined value. Suppose we are asked to construct

$$x' : \mu\ D\ i \qquad \text{such that} \qquad \mathit{fold}\ \phi\ x' \equiv j\ .$$

Instead of constructing x' directly and proving afterwards that the specification is satisfied, we can construct an ornamented object

$$x : \mu\ \lfloor \mathit{algOrn}\ D\ \phi \rfloor\ (i, j)$$

and set

$$x' = \mathit{forget}\ (\mathit{algOrn}\ D\ \phi)\ x\ :\ \mu\ D\ i\ .$$

Then the recomputation lemma says exactly that x' satisfies the specification. This construction method is central to the realisability interpretation we are proposing.

3. A realisability interpretation of ornamental-algebraic ornaments

From now on, we focus on what we might call *ornamental-algebraic ornaments*, i.e., algebraic ornaments induced by algebras that are themselves ornamental algebras; these can be given an intuitive interpretation, taking inspiration from the theory of *realisability*. In the Curry-Howard world, we are familiar with what it means for a proof term to *prove* a proposition, i.e., to inhabit a type — the term is related to the type by the typing meta-relation. Compare this with the realisability view, under which we say a term x' *realises* (instead of *proves*) a proposition φ when x' is related to φ by some relation defined in the language, traditionally written $x' \Vdash \varphi$. The predicate $\lambda x' \mapsto x' \Vdash \varphi$ is called the *realisability predicate* for φ, so saying that x' realises φ is equivalent to saying that x' satisfies the realisability predicate for φ. When $x' \Vdash \varphi$ holds, the term x' is called a *realiser* of the proposition φ. The relation $\_\Vdash\_$ being defined in the language, a proof of $x' \Vdash \varphi$ exists as a proof term, which we call a *realisability proof*. In his original realisability paper [9], Kleene hinted that a realiser is "an incomplete communication of a more specific statement," and a realisability proof provides "items as may be necessary to complete the communication." This is close to our interpretation: A realiser is an incomplete proof that can somehow be derived from a complete proof without losing the basic structure of the latter. A realisability predicate states what information needs to be supplied if we wish to augment an incomplete proof to a complete one, and a realisability proof provides the missing information.

For example, let us consider lists as realisers of the vector type. That is, lists are incomplete vectors, in the sense that a list is a vector whose length information is forgotten. To synthesise a vector of length n out of a list, we need to prove that the list has length n. One way to state this is to use the inductively defined relation

$$\begin{array}{l} \textbf{data}\ \mathsf{Length}\ \{A : \mathsf{Set}\}\ : \mathsf{Nat} \to \boxed{\mathsf{List}\ A} \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{nil}\ \ : \mathsf{Length}\ \mathsf{zero}\ \boxed{[\,]} \\ \quad \mathsf{cons}\ : \forall\, \{x\ n\ \boxed{xs}\} \to \\ \qquad\qquad \mathsf{Length}\ n\ \boxed{xs} \to \mathsf{Length}\ (\mathsf{suc}\ n)\ \boxed{(x :: xs)}\ . \end{array}$$

The predicate $\mathsf{Length}\ n$ on lists is the realisability predicate which, when satisfied by a list, states that the list can be upgraded to a vector of length n. That is, we can establish an isomorphism

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

which allows a list that can be proved to have length n to be converted to a vector of length n or vice versa. If we temporarily ignore the framed parts of Length, we see that it is just the vector type, so an inhabitant of $\mathsf{Length}\ n\ xs$ is actually a vector of length n whose type is indexed by the underlying list xs. Since the underlying list is computed by the forgetful map, we see that Length is the algebraic ornamentation of $\mathsf{Vec}\ A$ using the ornamental algebra from vectors to lists. The use of the ornament language here is a hint of datatype-genericity.

So let us go generic: An ornament $O : \mathsf{Orn}\ J\ e\ D$ of a description $D : \mathsf{Desc}\ I$ states how to augment the datatype $\mu\ D : I \to \mathsf{Set}$ to a richer datatype $\mu\ \lfloor O \rfloor : J \to \mathsf{Set}$, and induces a forgetful map $\mathit{forget}\ O : \mu\ \lfloor O \rfloor \Rightarrow \mu\ D \cdot e$. If we regard $\mu\ \lfloor O \rfloor$ as the *complete type*, then $\mu\ D$ is incomplete with respect to $\mu\ \lfloor O \rfloor$ and serves as the type of *potential* realisers of $\mu\ \lfloor O \rfloor$.[3] A complete object of type $\mu\ \lfloor O \rfloor\ j$ can be compressed by $\mathit{forget}\ O$ to an incomplete one of type $\mu\ D\ (e\ j)$ but retaining the basic structure. Conversely, given an incomplete object $x' : \mu\ D\ (e\ j)$, can we reconstruct a complete object $x : \mu\ \lfloor O \rfloor\ j$ such that x has the same basic structure as x', i.e., $\mathit{forget}\ O\ x \equiv x'$ is provable? This is exactly the scenario where the construction method supported by algebraically ornamented datatypes can be applied, since $\mathit{forget}\ O = \mathit{fold}\ (\mathit{ornAlg}\ O)$. So the answer is: Yes, if we can construct

$$r : \mu\ \lfloor \mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O) \rfloor\ (j, x')\ ,$$

then by setting

$$x = \mathit{forget}\ (\mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O))\ r\ :\ \mu\ \lfloor O \rfloor\ j$$

we are assured by the recomputation lemma that

$$\begin{array}{ll} & \mathit{forget}\ O\ x \\ \equiv & \mathit{fold}\ (\mathit{ornAlg}\ O)\ x \\ \equiv & \mathit{fold}\ (\mathit{ornAlg}\ O)\ (\mathit{forget}\ (\mathit{algOrn}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O))\ r) \\ \equiv & \quad \{\text{- }\ \mathit{recomputation}\ \lfloor O \rfloor\ (\mathit{ornAlg}\ O)\ r\ \text{ -}\} \\ & \mathsf{proj}_2\ (j, x') \\ \equiv & x'\ . \end{array}$$

[3] A $\mu\ D$ object is not necessarily a realiser of $\mu\ \lfloor O \rfloor$, as it may not satisfy the realisability predicate. We will nevertheless simply call $\mu\ D$ the *realiser type*, instead of the "potential realiser type."

The predicate $\lambda x' \mapsto \mu\ \lfloor algOrn\ \lfloor O \rfloor\ (ornAlg\ O) \rfloor\ (j, x')$ thus acts as the realisability predicate. Consequently we define

$$\begin{aligned}&rpOrn : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\ &\qquad \mathsf{Orn}\ (\Sigma\ J\ (\mu\ D \cdot e))\ \mathsf{proj}_1\ \lfloor O \rfloor \\ &rpOrn\ \ O\ =\ algOrn\ \lfloor O \rfloor\ (ornAlg\ O)\end{aligned}$$

and

$$\begin{aligned}&[\_]\_\Vdash\_ : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\} \to \\ &\qquad (j : J)\ (x' : \mu\ D\ (e\ j))\ (O : \mathsf{Orn}\ J\ e\ D) \to \mathsf{Set} \\ &[j]\ x' \Vdash O\ =\ \mu\ \lfloor rpOrn\ O \rfloor\ (j, x')\ .\end{aligned}$$

This is interpreted as the type of a proof that x' can be completed to yield an object of $\mu\ \lfloor O \rfloor$. We also define $x' \Vdash O = [\_]\ x' \Vdash O$ so the index j can be omitted when it can be inferred.

Since a realisability predicate is implemented as an algebraic ornamentation of the complete type, it is isomorphic to the complete type — more precisely, to be called a type a realisability predicate needs to be applied to a realiser, so the complete type is isomorphic to the dependent pair of the realiser type and the realisability predicate. The isomorphism can be nicely interpreted in terms of realisability: Given a complete object, a realiser can be obtained by applying *forget O* to the object, and the corresponding realisability proof is produced by

$$\begin{aligned}&\Re : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \\ &\qquad \{j\}\ (x : \mu\ \lfloor O \rfloor\ j) \to forget\ O\ x \Vdash O \\ &\Re\ \ O\ \ x\ =\ remember\ \lfloor O \rfloor\ (ornAlg\ O)\ x\ .\end{aligned}$$

We call this direction of the isomorphism the *realisability transformation*, because it helps to switch from the "proving" view to the "realising" view. The inverse transformation metaphorically combines a realiser and its realisability proof, whose computation depends only on the latter:

$$\begin{aligned}&\Re^{-1} : \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \\ &\qquad \{j\}\ (x' : \mu\ D\ (e\ j))\ (r : x' \Vdash O) \to \mu\ \lfloor O \rfloor\ j \\ &\Re^{-1}\ \ O\ \ x'\ \ r\ =\ forget\ (rpOrn\ O)\ r\ .\end{aligned}$$

That $\Re$ and $\Re^{-1}$ are indeed inverse to each other can be proven by *recomputation* and the fact that *forget* and *remember* are inverses. For example, one inverse property we will need is

$$\begin{aligned}&realiser\text{-}recovery : \\ &\quad \forall\ \{I\ J\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D) \to \\ &\quad (x' : \mu\ D\ (e\ j))\ (r : x' \Vdash O) \to forget\ O\ (\Re^{-1}\ O\ x'\ r) \equiv x' \\ &realiser\text{-}recovery\ O\ x'\ r = recomputation\ \lfloor O \rfloor\ (ornAlg\ O)\ r\ ,\end{aligned}$$

which says that the realiser extracted from a complete object synthesised from a realiser x' is just x' again.

To recap: By defining an ornament, we specify a complete type relative to a realiser type, and the corresponding realisability predicate can be immediately derived from that ornament. From this follow the realisability transformation and its inverse transformation that allow us to break a complete object into a realiser and a corresponding realisability proof, or recover a complete object from a realisability proof, which depends on a realiser.

Examples. Let us turn back to the example in which lists are viewed as realisers of the vector type, which arises from the ornament *VecO A*. The derived realisability predicate is

$$\begin{aligned}&Length : \forall\ \{A\} \to Nat \to List\ A \to \mathsf{Set} \\ &Length\ \ \{A\}\ \ n\ \ xs\ =\ [\mathsf{tt}, n]\ xs \Vdash VecO\ A\ ,\end{aligned}$$

which translates to the Length datatype given previously.

A slightly confusing but classic example is given by the ornament *ListO A*. The complete type specified by this ornament is *List A*, and the realiser type is *Nat* — a natural number is an incomplete list, with the elements missing. The derived realisability predicate $n \Vdash ListO\ A$ is just $Vec\ A\ n$, meaning that to augment a natural number n to a list of A's we need to supply a vector of type $Vec\ A\ n$, i.e., n elements of type A. It may seem strange at first that to construct a list, we end up constructing a vector, which is "heavier" than a list. But in fact we are asking not just for any list, but a list whose length is n. By constructing the list as a vector indexed by n, the requirement that the list constructed should have length n, i.e., that it has the same basic structure as n, is met by construction. A metaphor for this is that n is scaffolding to guide the construction of a list, and a vector is the finished construction still with the scaffold. To get the list constructed, we remove the scaffold by $\Re^{-1}$, i.e., *forget*.

For an example other than vectors, assume that a less-than-or-equal-to relation $\_\leq\_$ on natural numbers is suitably defined, and consider the following datatype for sorted lists of natural numbers indexed by a lower bound:

$$\begin{aligned}&\textbf{data}\ \mathsf{SList} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\ &\quad \mathsf{snil}\ \ \ : \forall\ \{b\} \to \mathsf{SList}\ b \\ &\quad \mathsf{scons}\ : (x : \mathsf{Nat}) \to \forall\ \{b\} \to b \leq x \to \mathsf{SList}\ x \to \mathsf{SList}\ b\ .\end{aligned}$$

Coding sorted lists as an ornamentation of lists,

$$\begin{aligned}&SListO\ \ :\ \ \mathsf{Orn}\ Nat\ !\ \lfloor List\ Nat \rfloor \\ &SListO\ =\ \sigma\ \mathsf{Bool}\ (false \mapsto \Delta\ Nat\ (\lambda b \mapsto \mathsf{say}\ (\mathsf{ok}\ b)) \\ &\qquad\qquad true \mapsto\ \sigma\ Nat\ (\lambda x \mapsto \\ &\qquad\qquad\qquad \Delta\ Nat\ (\lambda b \mapsto \Delta\ (b \leq x)\ (\lambda\_ \mapsto \\ &\qquad\qquad\qquad\quad \mathsf{ask}\ (\mathsf{ok}\ x) * \mathsf{say}\ (\mathsf{ok}\ b))))))\ ,\end{aligned}$$

where $! = \lambda\_ \mapsto \mathsf{tt} : \forall\ \{A\} \to A \to \top$, we obtain $SList = \mu\ \lfloor SListO \rfloor$. The derived realisability predicate is

$$\begin{aligned}&Sorted : Nat \to List\ Nat \to \mathsf{Set} \\ &Sorted\ \ n\ \ xs\ =\ [n]\ xs \Vdash SListO\ Nat\ ,\end{aligned}$$

which translates to

$$\begin{aligned}&\textbf{data}\ \mathsf{Sorted} : \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\ &\quad \mathsf{nil}\ \ \ : \forall\ \{b\} \to \mathsf{Sorted}\ b\ [\,] \\ &\quad \mathsf{cons}\ : \forall\ \{x\ b\} \to b \leq x \to \\ &\qquad\qquad \forall\ \{xs\} \to \mathsf{Sorted}\ x\ xs \to \mathsf{Sorted}\ b\ (x :: xs)\ .\end{aligned}$$

It is an inductively defined predicate stating that a list is sorted and bounded below by a number. If we can prove that a list satisfies this predicate, then the list can be cast as a sorted list bounded below.

For an example other than lists, recall the finite numbers presented in Section 1, which can be coded as an ornamentation of natural numbers:

$$\begin{aligned}&FinO : \mathsf{Orn}\ Nat\ !\ NatD \\ &FinO = \\ &\quad \sigma\ \mathsf{Bool}\ (false \mapsto \Delta\ Nat\ (\lambda n \mapsto \mathsf{say}\ (\mathsf{ok}\ (suc\ n))) \\ &\qquad\qquad true \mapsto\ \Delta\ Nat\ (\lambda n \mapsto \mathsf{ask}\ (\mathsf{ok}\ n) * \mathsf{say}\ (\mathsf{ok}\ (suc\ n))))\ .\end{aligned}$$

The decoded type of finite numbers is thus $Fin = \mu\ \lfloor FinO \rfloor$. The derived realisability predicate translates to the greater-than relation also presented in Section 1 — to say a natural number n is a finite number bounded by m, what we need to prove is exactly $m > n$.

Realisability predicates for algebraic ornaments. The example regarding lists as realisers of the vector type may have made the reader feel uneasy — the derived realisability predicate *Length* looks rather heavyweight. Given that an algebraic ornament does not add extra information to a datatype, shouldn't the realisability predicate be more lightweight and sometimes even trivially satisfiable? Indeed, the realisability predicate for an algebraic ornament should simply amount to an equality, e.g., $length\ xs \equiv n$ for the ornament *VecO A* instead of *Length n xs*, and this can be proved generically.

For one direction, we wish to prove

$$\begin{aligned}&AOE\ :\ \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\qquad \{ij : \Sigma\ I\ J\}\ \{x' : \mu\ D\ (\mathsf{proj}_1\ ij)\} \to \\ &\qquad (r : [ij]\ x' \Vdash algOrn\ D\ \phi) \to fold\ \phi\ x' \equiv \mathsf{proj}_2\ ij\ .\end{aligned}$$

Note that the realiser x' is ornamentally two levels away from the realisability proof r, but since the two ornaments involved are both algebraic, x' and r really are isomorphic. The goal type looks quite similar to the conclusion of the recomputation lemma at the first level, so we try to apply *recomputation* to a complete object, the natural choice being $\Re^{-1}\ (algOrn\ D\ \phi)\ x'\ r$. We supply the proof term

$$recomputation\ D\ \phi\ (\Re^{-1}\ (algOrn\ D\ \phi)\ x'\ r) \tag{1}$$

as the result, whose type

$$fold\ \phi\ (forget\ (algOrn\ D\ \phi)\ (\Re^{-1}\ (algOrn\ D\ \phi)\ x'\ r)) \equiv \mathsf{proj}_2\ ij$$

requires a bit of tweaking, though: The argument to $fold\ \phi$ should be just x' to match the goal type, which is achieved by rewriting with

$$realiser\text{-}recovery\ (algOrn\ D\ \phi)\ x'\ r\ . \tag{2}$$

In Agda, we first use **with** to put the term (1) into the context and then **rewrite** the context with (2) before delivering the term left in the context as the result, whose type has been appropriately rewritten. This programming pattern will be used a lot.

The other direction requires us to prove

$$\begin{aligned} AOE^{-1}\ :\ &\forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\{ij : \Sigma\ I\ J\}\ \{x' : \mu\ D\ (\mathsf{proj}_1\ ij)\} \rightarrow \\ &fold\ \phi\ x' \equiv \mathsf{proj}_2\ ij \rightarrow [ij]\ x' \Vdash algOrn\ D\ \phi\ . \end{aligned}$$

Promoting x' two levels up should do the job, so we give the term

$$\Re\ (algOrn\ D\ \phi)\ (remember\ D\ \phi\ x')$$

as the result after tweaking its type, which is originally

$$\begin{aligned} &[\mathsf{proj}_1\ ij, fold\ \phi\ x'] \\ &\quad forget\ (algOrn\ D\ \phi)\ (remember\ D\ \phi\ x') \Vdash algOrn\ D\ \phi\ . \end{aligned}$$

Since *forget* is a left inverse to *remember* and we have an assumption $fold\ \phi\ x' \equiv \mathsf{proj}_2\ ij$, the type can be rewritten as our goal.

Incidentally, implementation of *remember* and *recomputation* can be made symmetric under the realisability view, i.e., if the combinators $\Re$, $\Re^{-1}$, *AOE*, and AOE^{-1} are taken as primitives. First consider *remember*: To promote $x' : \mu\ D\ i$ to an object of type

$$\mu\ \lfloor algOrn\ D\ \phi \rfloor\ (i, fold\ \phi\ x')\ ,$$

we apply the inverse realisability transformation $\Re^{-1}$ to x' and a corresponding realisability proof, which can simply be a proof of $fold\ \phi\ x' \equiv fold\ \phi\ x'$ because of AOE^{-1}.

$$\begin{aligned} &remember\ : \\ &\quad \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\quad \{i\}\ (x' : \mu\ D\ i) \rightarrow \mu\ \lfloor algOrn\ D\ \phi \rfloor\ (i, fold\ \phi\ x') \\ &remember\ D\ \phi\ x' = \Re^{-1}\ (algOrn\ D\ \phi)\ x'\ (AOE^{-1}\ D\ \phi\ \mathsf{refl}) \end{aligned}$$

As for *recomputation*, we are given a complete object x and asked to produce the equality form of a realisability proof, which we can easily obtain by applying the realisability transformation $\Re$ to x and then resorting to *AOE*.

$$\begin{aligned} &recomputation\ : \\ &\quad \forall\ \{I\ J\}\ (D : \mathsf{Desc}\ I)\ (\phi : [\![D]\!]\ J \Rightarrow J) \\ &\quad \{ij : \Sigma\ I\ J\}\ (x : \mu\ \lfloor algOrn\ D\ \phi \rfloor\ ij) \rightarrow \\ &\qquad fold\ \phi\ (forget\ (algOrn\ D\ \phi)\ x) \equiv \mathsf{proj}_2\ ij \\ &recomputation\ D\ \phi\ x = AOE\ D\ \phi\ (\Re\ (algOrn\ D\ \phi)\ x) \end{aligned}$$

We see that *remember* is $\Re^{-1}$ whose argument is produced by AOE^{-1}, while *recomputation* is $\Re$ whose result is modified by *AOE*. Hence *recomputation* and *remember* are actually the realisability transformation and the inverse transformation, specialised for algebraic ornaments.

Function upgrade. After we define an ornament to get a fancier type, naturally we want to port functions working on the original type to the fancier type. For example, we should be able to upgrade list append to vector append. Our strategy is based on realisers and realisability predicates for *function types*. In type theory, a proof of an implication $\varphi \rightarrow \psi$ takes a proof of φ to a proof of ψ, while in the realisability world the role of proofs are taken by realisers, so a realiser of $\varphi \rightarrow \psi$ takes a realiser of φ to a realiser of ψ, i.e., it is a function $f' : \varphi' \rightarrow \psi'$ where φ' and ψ' are the type of potential realisers of φ and ψ. But in order to justify that f' really takes realisers to realisers, we need to prove that when an input $x' : \varphi'$ is a realiser, i.e., we are given a proof of $x' \Vdash \varphi$, the output $f'\ x' : \psi'$ is also a realiser, i.e., we can produce a proof of $f'\ x' \Vdash \psi$. This justification is thus a proof of type

$$(x' : \varphi') \rightarrow x' \Vdash \varphi \rightarrow f'\ x' \Vdash \psi\ ,$$

which is defined to be the realisability predicate for $\varphi \rightarrow \psi$. Back in the context of ornaments, this suggests that to upgrade a function, we can consider it as a realiser of a function type between ornamented types, and obtain a complete function by supplying a suitable realisability proof.

For the example of upgrading list append to vector append, our goal is to write the append function for vectors

$$vappend : \forall\ \{A\ m\ n\} \rightarrow Vec\ A\ m \rightarrow Vec\ A\ n \rightarrow Vec\ A\ (m+n)$$

in terms of list append

$$\_{+\!\!+}\_ : \forall\ \{A\} \rightarrow List\ A \rightarrow List\ A \rightarrow List\ A\ .$$

Given $xs : Vec\ A\ m$ and $ys : Vec\ A\ n$, to produce a vector of type $Vec\ A\ (m+n)$, we invoke $\Re^{-1}$ and thereby split the goal into two parts — the realiser and the realisability proof. The realiser, which is a list, is obtained by extracting the two underlying lists $xs' = forget\ (VecO\ A)\ xs$ and $ys' = forget\ (VecO\ A)\ ys$ and appending them. For the realisability proof, because of AOE^{-1}, the proof obligation is reduced to the equality

$$length\ (xs' +\!\!+ ys') \equiv m+n\ .$$

We know *length* is a list homomorphism, i.e.,

$$length\ (xs' +\!\!+ ys') \equiv length\ xs' + length\ ys' \quad \text{for all } xs' \text{ and } ys'\ .$$

The type of the realisability proof for list append is merely a restatement of the fact above:

$$\begin{aligned} &append\text{-}length\ : \\ &\quad \forall\ \{A\}\ (xs'\ ys' : List\ A)\ \{m\ n\} \rightarrow \\ &\quad length\ xs' \equiv m \rightarrow length\ ys' \equiv n \rightarrow length\ (xs' +\!\!+ ys') \equiv m+n\ . \end{aligned}$$

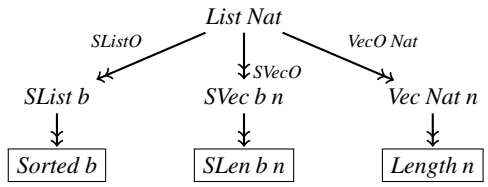
And the two equality premises of *append-length* are discharged by applying $\Re$ and *AOE* to *xs* and *ys*. The whole Agda translation is shown below.

$$\begin{aligned} &vappend : \forall\ \{A\ m\ n\} \rightarrow Vec\ A\ m \rightarrow Vec\ A\ n \rightarrow Vec\ A\ (m+n) \\ &vappend\ \{A\}\ xs\ ys = \Re^{-1}\ (VecO\ A)\ (xs' +\!\!+ ys') \\ &\quad (AOE^{-1}\ \lfloor ListO\ A \rfloor\ \phi\ (append\text{-}length\ xs'\ ys'\ eq_1\ eq_2)) \\ &\quad \textbf{where}\ xs' = forget\ (VecO\ A)\ xs \\ &\qquad\qquad ys' = forget\ (VecO\ A)\ ys \\ &\qquad\qquad \phi = ornAlg\ (ListO\ A) \\ &\qquad\qquad eq_1 = AOE\ \lfloor ListO\ A \rfloor\ \phi\ (\Re\ (VecO\ A)\ xs) \\ &\qquad\qquad eq_2 = AOE\ \lfloor ListO\ A \rfloor\ \phi\ (\Re\ (VecO\ A)\ ys) \end{aligned}$$

4. A first step towards ornament composition

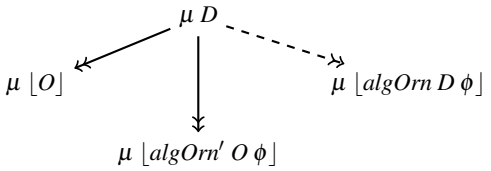
The inverse realisability transformation combines a realisability proof with a realiser to get a complete object. For example, we combine length information with a list to get a vector, or we combine a proof of the *Sorted* predicate with a list to get a sorted list. It is then natural to ask: Can we combine *both* the length information and the sortedness proof with a list, to get a sorted vector?

To clarify, the datatypes involved are shown in the following diagram, ornaments drawn as double-headed arrows and the realisability predicates framed. It can be read as "the datatype *List Nat* is revised to the datatype *SList b* by the ornament *SListO*" and so on.



We know that to promote a list *xs* to a sorted vector, we need to provide a realisability proof of type *SLen b n xs*, but what we are given are proofs of *Sorted b xs* and *Length n xs*. Nevertheless, intuitively we see that *Sorted b xs* × *Length n xs* is isomorphic to *SLen b n xs*, i.e., the realisability predicate for the ornament *SVecO* is the composition (pointwise conjunction) of the realisability predicates for *SListO* and *VecO Nat*. Since realisability predicates are derived from ornaments, we are led to seeking a way of regarding *SVecO* as the composition of *SListO* and *VecO Nat*. Our hypothesis, then, is that the realisability predicate for a composite ornament amounts to the composition (pointwise conjunction) of the realisability predicates for the component ornaments.

As an initial experiment, in this paper we consider only composition of two ornaments one of which is algebraic, which has a simpler implementation. Let $D : \mathsf{Desc}\ I$, $O : \mathsf{Orn}\ J\ e\ D$, and $\phi : [\![D]\!]\ K \Rightarrow K$. The composition of O and $algOrn\ D\ \phi$ will be called $algOrn'\ O\ \phi$. The datatypes involved are shown in the following diagram. We omit the names of ornaments on the arrows that represent them, because the names are shown in the datatypes at the end of the arrows. A dashed arrow indicates an algebraic ornament.



The function $algOrn'$ does the same thing as $algOrn$ except that it works *on an ornament* — $algOrn'\ O\ \phi$ patches O algebraically so the resulting ornament on D has new indices which are the result of folding with ϕ. We call it an *algebraic ornament-ornament*. (Fortunately this rather ugly name will appear only once more.)

$$
\begin{aligned}
&algOrn' : \{I\ J : \mathsf{Set}\}\ \{K : I \to \mathsf{Set}\}\ \{e : J \to I\}\ \{D : \mathsf{Desc}\ I\}\\
&\qquad (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K) \to\\
&\qquad \mathsf{Orn}\ (\Sigma\ J\ (K \cdot e))\ (e \cdot \mathsf{proj}_1)\ D\\
&algOrn'\ (\mathsf{say}\ (\mathsf{ok}\ j))\ \phi\ =\ \mathsf{say}\ (\mathsf{ok}\ (j, \phi\ \mathsf{refl}))\\
&algOrn'\ (\sigma\ S\ O)\qquad \phi\ =\ \sigma\ S\ \lambda s \mapsto algOrn'\ (O\ s)\ (\Lambda\ \phi\ s)\\
&algOrn'\ \{K = K\}\ \{e\}\ (\mathsf{ask}\ (\mathsf{ok}\ j) * O)\ \phi =\\
&\qquad \Delta\ (K\ (e\ j))\ \lambda k \mapsto \mathsf{ask}\ (\mathsf{ok}\ (j,k)) * algOrn'\ O\ (\Lambda\ \phi\ k)\\
&algOrn'\ (\Delta\ S\ O)\qquad \phi\ =\ \Delta\ S\ \lambda s \mapsto algOrn'\ (O\ s)\ \phi\ .
\end{aligned}
$$

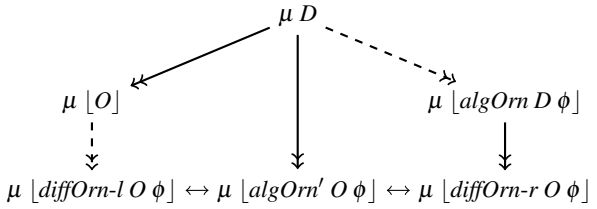
Each of the three ornaments appearing in the diagram induces its own realisability predicate, and we are going to show that a realisability proof for $algOrn'\ O\ \phi$ can be projected to a realisability proof for O or for $algOrn\ D\ \phi$, or synthesised by integrating realisability proofs for O and $algOrn\ D\ \phi$.

Projections. First we deal with the left and right projection,

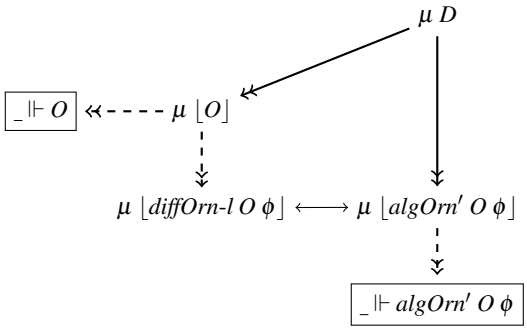
$$
\begin{aligned}
&project\text{-}l :\\
&\quad \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K)\\
&\quad \{jk : \Sigma\ J\ (K \cdot e)\}\ \{x' : \mu\ D\ (e\ (\mathsf{proj}_1\ jk))\} \to\\
&\quad (r : [jk]\ x' \Vdash algOrn'\ O\ \phi) \to [\mathsf{proj}_1\ jk]\ x' \Vdash O
\end{aligned}
$$

$$
\begin{aligned}
&project\text{-}r :\\
&\quad \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K)\\
&\quad \{jk : \Sigma\ J\ (K \cdot e)\}\ \{x' : \mu\ D\ (e\ (\mathsf{proj}_1\ jk))\} \to\\
&\quad (r : [jk]\ x' \Vdash algOrn'\ O\ \phi) \to [(e \times id)\ jk]\ x' \Vdash algOrn\ D\ \phi\ ,
\end{aligned}
$$

where $\_\times\_$ is overloaded to mean $(f \times g)\ (x, y) = (f\ x, g\ y)$. It is difficult to prove the projections directly by generic induction on r. Rather, we will first complete the ornament diagram by defining two *difference ornaments* from which the decoded datatypes are isomorphic to $\mu\ \lfloor algOrn'\ O\ \phi \rfloor$ (the isomorphisms are shown as two-way arrows below),



and then route a realisability proof through the appropriate forgetful maps, isomorphisms, and remembering maps to get the proof we want. Let us look at the left part of the completed diagram, to which the induced realisability predicates have been added.



The left difference ornament is an algebraic ornament defined by

$$
\begin{aligned}
&diffOrn\text{-}l : \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\\
&\qquad (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K) \to\\
&\qquad \mathsf{Orn}\ (\Sigma\ J\ (K \cdot e))\ \mathsf{proj}_1\ \lfloor O \rfloor\\
&diffOrn\text{-}l\ O\ \phi\ =\ algOrn\ \lfloor O \rfloor\ (\phi \cdot erase\ O)\ .
\end{aligned}
$$

One direction of the isomorphism between $\mu\ \lfloor diffOrn\text{-}l\ O\ \phi \rfloor$ and $\mu\ \lfloor algOrn'\ O\ \phi \rfloor$ is iso_1,

$$
\begin{aligned}
&iso_1 : \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\\
&\qquad (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K) \to\\
&\qquad \mu\ \lfloor algOrn'\ O\ \phi \rfloor \Rightarrow \mu\ \lfloor diffOrn\text{-}l\ O\ \phi \rfloor\\
&iso_1\ O\ \phi\ =\ fold\ (\langle \_ \rangle \cdot iso_1\text{-}cast\ O\ \phi)\ ,
\end{aligned}
$$

where iso_1-*cast* is a polymorphic restructuring map like *erase*, which is actually just an identity map.

$$
\begin{aligned}
&iso_1\text{-}cast : \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\\
&\qquad (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K)\ \{X\} \to\\
&\qquad [\![\lfloor algOrn'\ O\ \phi \rfloor]\!]\ X \Rightarrow [\![\lfloor diffOrn\text{-}l\ O\ \phi \rfloor]\!]\ X\\
&iso_1\text{-}cast\ (\mathsf{say}\ (\mathsf{ok}\ j))\ \phi\ \mathsf{refl} = \mathsf{refl}\\
&iso_1\text{-}cast\ (\sigma\ S\ O)\ \phi\ (s, xs) = s, iso_1\text{-}cast\ (O\ s)\ (\Lambda\ \phi\ s)\ xs\\
&iso_1\text{-}cast\ (\mathsf{ask}\ (\mathsf{ok}\ j) * O)\ \phi\ (k, x, xs) = k, x,\\
&\qquad\qquad iso_1\text{-}cast\ O\ (\Lambda\ \phi\ k)\ xs\\
&iso_1\text{-}cast\ (\Delta\ S\ O)\ \phi\ (s, xs) = s, iso_1\text{-}cast\ (O\ s)\ \phi\ xs
\end{aligned}
$$

The other direction of the isomorphism, iso_2, has the same implementation. Each ornament induces a forgetful map, and additionally a remembering map if it is algebraic, as shown in Figure 1.

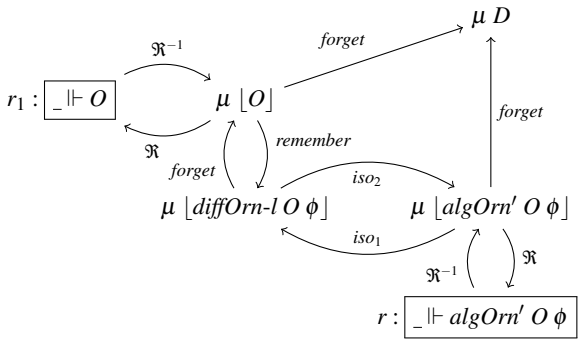


Figure 1. (Commutative) diagram of ornament-induced maps and the isomorphism maps iso_1 and iso_2.

In the diagram there is a path of maps along which we can take a realisability proof for $algOrn'\ O\ \phi$ to one for O: Starting from a composite realisability proof

$$r : [jk]\ x' \Vdash algOrn'\ O\ \phi\ ,$$

the term

$$r_1 = \Re\ O\ (forget\ (diffOrn\text{-}l\ O\ \phi)\ (iso_1\ O\ \phi\ (\Re^{-1}\ (algOrn'\ O\ \phi)\ x'\ r)))$$

is the desired left-component realisability proof, but its type again needs tweaking. Its original type is

$$[\mathsf{proj}_1\ jk]\ forget\ O\ (forget\ (diffOrn\text{-}l\ O\ \phi)\ (iso_1\ O\ \phi\ (\Re^{-1}\ (algOrn'\ O\ \phi)\ x'\ r))) \Vdash O\ ,$$

while our goal type is

$$[\mathsf{proj}_1\ jk]\ x' \Vdash O\ .$$

The composition of the two *forget*s and iso_1, however, can be reduced to just one big $forget\ (algOrn'\ O\ \phi)$. This can be proved by two applications of fold fusion [5], and ultimately reduces to naturality [16] of the underlying restructuring maps — *erase* and iso_1-*cast* — and the fact that

$$erase\ O\ (erase\ (diffOrn\text{-}l\ O\ \phi)\ (iso_1\text{-}cast\ O\ \phi\ xs)) \equiv erase\ (algOrn'\ O\ \phi)\ xs$$

holds for all xs, which can be easily proved by induction on O. Thus the type we are left with is

$$[\mathsf{proj}_1\ jk]\ forget\ (algOrn'\ O\ \phi)\ (\Re^{-1}\ (algOrn'\ O\ \phi)\ x'\ r) \Vdash O\ ,$$

and *realiser-recovery* says that the realiser is just x'. Thus we reach our goal type and finish the implementation of the left projection.

As for the right projection, after defining the right difference ornament,

$$\begin{aligned}
&diffOrn\text{-}r : \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\\
&\qquad (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K) \rightarrow\\
&\qquad \mathsf{Orn}\ (\Sigma\ J\ (K \cdot e))\ (e \times id)\ \lfloor algOrn\ D\ \phi \rfloor\\
&diffOrn\text{-}r\ (\mathsf{say}\ (\mathsf{ok}\ j))\ \ \phi\ =\ \mathsf{say}\ (\mathsf{ok}\ (j, \mathsf{refl}))\\
&diffOrn\text{-}r\ (\sigma\ S\ O)\qquad \phi\ =\ \sigma\ S\ \lambda s \mapsto diffOrn\text{-}r\ (O\ s)\ (\Lambda\ \phi\ s)\\
&diffOrn\text{-}r\ \{K = K\}\ \{e\}\ (\mathsf{ask}\ (\mathsf{ok}\ j) * O)\ \phi =\\
&\qquad \sigma\ (K\ (e\ j))\ \lambda k \mapsto \mathsf{ask}\ (\mathsf{ok}\ (j, k)) * diffOrn\text{-}r\ O\ (\Lambda\ \phi\ k)\\
&diffOrn\text{-}r\ (\Delta\ S\ O)\qquad \phi\ =\ \Delta\ S\ \lambda s \mapsto diffOrn\text{-}r\ (O\ s)\ \phi\ ,
\end{aligned}$$

the implementation is completely symmetric and is omitted here.

Integration. Now we look at integration:

$$\begin{aligned}
&integrate :\\
&\quad \forall\ \{I\ J\ K\ e\}\ \{D : \mathsf{Desc}\ I\}\ (O : \mathsf{Orn}\ J\ e\ D)\ (\phi : [\![D]\!]\ K \Rightarrow K)\\
&\quad \{jk\}\ \{x' : \mu\ D\ (e\ (\mathsf{proj}_1\ jk))\} \rightarrow\\
&\quad (r_1 : [\mathsf{proj}_1\ jk]\ x' \Vdash O)\ (r_2 : [(e \times id)\ jk]\ x' \Vdash algOrn\ D\ \phi) \rightarrow\\
&\qquad [jk]\ x' \Vdash algOrn'\ O\ \phi\ .
\end{aligned}$$

Had we considered general ornament composition, integration would have been much harder to implement, because ingredients from both component realisability proofs are essential and really need to be integrated by hard work. But since we are considering algebraic ornament-ornaments, the left difference ornament is algebraic and thus induces a remembering map, completing a path of maps along which we can smuggle a realisability proof for O as one for $algOrn'\ O\ \phi$ — again see Figure 1. (A realisability proof for $algOrn\ D\ \phi$ is nevertheless still needed, which provides information about the index, as we will see later.) Starting from the left-component realisability proof

$$r_1 : [\mathsf{proj}_1\ jk]\ x' \Vdash O\ ,$$

the composite realisability proof we deliver is

$$r = \Re\ (algOrn'\ O\ \phi)\ (iso_2\ O\ \phi\ (remember\ \lfloor O \rfloor\ (\phi \cdot erase\ O)\ (\Re^{-1}\ O\ x'\ r_1)))\ ,$$

which has type

$$\begin{aligned}
&[\mathsf{proj}_1\ jk,\ fold\ (\phi \cdot erase\ O)\ (\Re^{-1}\ O\ x'\ r_1)]\\
&forget\ (algOrn'\ O\ \phi)\ (iso_2\ O\ \phi\\
&(remember\ \lfloor O \rfloor\ (\phi \cdot erase\ O)\ (\Re^{-1}\ O\ x'\ r_1))) \Vdash algOrn'\ O\ \phi\ ,
\end{aligned}$$

while our goal type is

$$[jk]\ x' \Vdash algOrn'\ O\ \phi\ .$$

Comparing the two types, we see that we need to establish two equalities,

$$fold\ (\phi \cdot erase\ O)\ (\Re^{-1}\ O\ x'\ r_1) \equiv \mathsf{proj}_2\ jk \tag{3}$$

and

$$forget\ (algOrn'\ O\ \phi)\ (iso_2\ O\ \phi\ (remember\ \lfloor O \rfloor\ (\phi \cdot erase\ O)\ (\Re^{-1}\ O\ x'\ r_1))) \equiv x'\ . \tag{4}$$

The left-hand side of the first equality (3) looks like the left-hand side of *realiser-recovery*, but instead of $fold\ (\phi \cdot erase\ O)$ what we wish to see is $forget\ O$. Nevertheless, we see that $fold\ (\phi \cdot erase\ O)$ is just $fold\ \phi$ lifted to work with $\mu\ \lfloor O \rfloor$, so we can perform fission [8] — the conceptual opposite of fusion — by proving that

$$fold\ (\phi \cdot erase\ O)\ x \equiv fold\ \phi\ (forget\ O\ x) \qquad \text{for all } x\ .$$

Hence the *forget* O coming out of the fission cancels out the $\Re^{-1}$ by *realiser-recovery*, reducing (3) to

$$fold\ \phi\ x' \equiv \mathsf{proj}_2\ jk\ . \tag{5}$$

This is where we need a realisability proof for $algOrn\ D\ \phi$. In the beginning we were also given the right-component realisability proof

$$r_2 : [(e \times id)\ jk]\ x' \Vdash algOrn\ D\ \phi\ .$$

Notice that r_2 is a realisability proof for an *algebraic* ornament, so it can be transformed by *AOE* to a proof of an equality, which is exactly (5). So the first equality is successfully discharged. As for the second equality (4), we perform fission again to exchange the big $forget\ (algOrn'\ O\ \phi)$ composed with iso_2 for two smaller *forget*s, one of which cancels out the *remember*. The equality is

thus reduced to

$$forget\ O\ (\mathfrak{R}^{-1}\ O\ x'\ r_1) \equiv x' \ ,$$

which is just an instance of *realiser-recovery*.

Example. Consider the function

$$\begin{array}{l} insert : Nat \to List\ Nat \to List\ Nat \\ insert\ y\ \langle \mathsf{false}, \mathsf{refl} \rangle = y :: [] \\ insert\ y\ \langle \mathsf{true}, x, xs, \mathsf{refl} \rangle\ \mathbf{with}\ y \leq_? x \\ \ldots \qquad\qquad\qquad\qquad \mid \mathsf{yes}\ \_ = y :: x :: xs \\ \ldots \qquad\qquad\qquad\qquad \mid \mathsf{no}\ \ \_ = x :: insert\ y\ xs \ , \end{array}$$

which is used, for example, in insertion sort. (The function $\_\leq_?\_$ compares two natural numbers, returning as a result either $\mathsf{yes}\ eq$ or $\mathsf{no}\ neq$ where eq and neq are proof terms justifying the result. Neither of the two proof terms is used in this basic version of *insert*, however.) We know that $insert\ y\ xs$ has one more element than xs, i.e., we can prove

$$\begin{array}{l} insert\text{-}length : \\ \quad \forall\ y\ xs\ \{n\} \to length\ xs \equiv n \to length\ (insert\ y\ xs) \equiv suc\ n \ . \end{array}$$

This is the realisability proof for upgrading *insert* to work with vectors, i.e., to the function

$$vinsert : Nat \to \forall\ \{n\} \to Vec\ Nat\ n \to Vec\ Nat\ (suc\ n) \ .$$

Also we know that *insert* produces a sorted list if the input list is sorted, i.e., we can prove

$$\begin{array}{l} insert\text{-}sorted : \\ \quad \forall\ y\ xs\ \{b\} \to Sorted\ b\ xs \to Sorted\ (b \sqcap y)\ (insert\ y\ xs) \end{array}$$

where $b \sqcap y$ is the minimum of b and y. Again this serves as a realisability proof for upgrading *insert* to work with sorted lists, i.e., to the function

$$sinsert : (y : Nat) \to \forall\ \{b\} \to SList\ b \to SList\ (b \sqcap y) \ .$$

Now suppose we wish to upgrade it to work with sorted vectors,

$$\begin{array}{l} \mathbf{data}\ \mathsf{SVec} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\ \mathbf{where} \\ \quad \mathsf{nil}\ \ : \forall\ \{b\} \to \mathsf{SVec}\ b\ \mathsf{zero} \\ \quad \mathsf{cons}\ : (x : \mathsf{Nat}) \to \forall\ \{b\} \to b \leq x \to \\ \qquad\qquad \forall\ \{n\} \to \mathsf{SVec}\ x\ n \to \mathsf{SVec}\ b\ (\mathsf{suc}\ n) \ , \end{array}$$

which is described by the ornament

$$\begin{array}{l} SVecO\ \ :\ \ \mathsf{Orn}\ (Nat \times Nat)\ !\ \lfloor ListO\ Nat \rfloor \\ SVecO\ =\ algOrn'\ SListO\ (ornAlg\ (ListO\ Nat)) \ . \end{array}$$

This time, however, we do not need to prove repetitively and monolithically that $insert\ y\ xs$ is sorted and has length $suc\ n$ if xs is sorted and has length n; instead, we can reuse *insert-length* and *insert-sorted* with the help of *project-l*, *project-r*, and *integrate*. The function we wish to write is

$$\begin{array}{l} svinsert : \\ \quad (y : Nat) \to \forall\ \{b\ n\} \to SVec\ b\ n \to SVec\ (b \sqcap y)\ (suc\ n) \ . \end{array}$$

Assume that $y : Nat$ and $xs : SVec\ b\ n$ are given. We invoke the inverse realisability transformation and supply $insert\ y\ xs'$, where $xs' = forget\ SVecO\ xs$, as the realiser, and we need to produce a corresponding realisability proof of type $insert\ y\ xs' \Vdash SVecO$ from a realisability proof of type $xs' \Vdash SVecO$. Since $SVecO$ is a composite ornament, we can break the given composite realisability proof into two component proofs with *project-l* and *project-r*, use them to build two required component proofs independently, and *integrate* the two independently built proofs to get the required composite proof. The program is shown below.

$$\begin{array}{l} svinsert : (y : Nat) \to \forall\ \{b\ n\} \to SVec\ b\ n \to SVec\ (b \sqcap y)\ (suc\ n) \\ svinsert\ y\ xs\ =\ \mathfrak{R}^{-1}\ SVecO\ (insert\ y\ xs') \\ \quad (integrate\ SListO\ \phi \\ \quad\quad (insert\text{-}sorted\ y\ xs'\ r_1) \\ \quad\quad (AOE^{-1}\ \lfloor ListO\ Nat \rfloor\ \phi \\ \quad\quad\quad (insert\text{-}length\ y\ xs'\ (AOE\ \lfloor ListO\ Nat \rfloor\ \phi\ r_2)))) \\ \quad \mathbf{where}\ xs' = forget\ SVecO\ xs \\ \qquad\qquad \phi = ornAlg\ (ListO\ Nat) \\ \qquad\qquad r = \mathfrak{R}\ SVecO\ xs \\ \qquad\qquad r_1 = project\text{-}l\ SListO\ \phi\ r \\ \qquad\qquad r_2 = project\text{-}r\ SListO\ \phi\ r \end{array}$$

5. Discussion

The realisability interpretation in fact works for *general* algebraic ornaments, ornamental-algebraic ornaments being a special case: Given a description $D : \mathsf{Desc}\ I$ and an algebra $\phi : [\![D]\!]\ J \Rightarrow J$, the type $\mu\ D$ is interpreted as the complete type, J as the realiser type, and $\mu\ \lfloor algOrn\ D\ \phi \rfloor$ as the realisability predicate. Assuming that $x : \mu\ D\ i$ is a complete object, the type of *remember* says that $fold\ \phi\ x : J\ i$ satisfies the realisability predicate, so *remember* is the realisability transformation, while the inverse transformation is *forget*. $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ are just *remember* and *forget* specialised for ornamental algebras. The reason we introduced the realisability transformation based on ornaments instead of algebras is that ultimately we use the transformation to talk about ornament composition. It is convenient to have the intuition that every ornament expresses the relationship between a realiser type and a complete type and induces a corresponding realisability predicate. Subsequently, composing ornaments gives rise to a new and richer complete type, and the induced realisability predicate can be decomposed into realisability predicates for the component ornaments. Algebra-based interpretation does not offer this intuition, because algebras do not compose: For example, we can fold both a list and a tree to a natural number, say computing the number of elements, but it is not obvious what composite datatype would arise in this situation.

More importantly, introducing the realisability interpretation in terms of ornamental-algebraic ornaments brings out the correspondence between internalism and externalism regarding constraint composition. Under the realisability view, data and constraints are separated into realisers and realisability predicates. This is exactly externalism — realisers do not carry with them proofs that they are indeed realisers. Multiple constraints simply correspond to multiple realisability predicates applied to the same piece of data. For internalism, constraints are encoded in ornaments, and to express multiple constraints we use ornament composition. The realisability transformation points out the correspondence between the two different ways of expressing constraints — ornaments for internalism and realisability predicates for externalism: An ornament *induces* a realisability predicate, which is the manifestation of the ornament in the world of decoded datatypes, and moreover, composition of realisability predicates mirrors composition of ornaments. A bridge is thus formed between externalism and internalism, and subsequently, externalist modularity is brought into internalist datatypes.

It is worth noting that upgrading a function using the realisability transformation does not really exempt us from reimplementing the logic. For example, when we upgrade *insert* to work with sorted lists, the realisability proof we need to supply is *insert-sorted*, which takes one *Sorted*ness proof and produces another. *Sorted* being isomorphic to *SList*, implementing *insert-sorted* is not so different from reimplementing *insert* for sorted lists. So what is the difference? Let us temporarily change our perspective and consider how we might synthesise *svinsert* from *sinsert* and *vinsert*, without the help of the realisability transformation. We would get a sorted list and a vector from the input sorted vector, feed them to *sinsert*

and *vinsert* separately, and combine the outputs to get a sorted vector as the final result. The main obstacle is that we cannot freely integrate a sorted list with a vector to get a sorted vector, because the underlying list of the sorted list may not be the same as that of the vector. If we are able to guarantee that the sorted list and the vector have the same underlying list, however, then the integration goes through, but it is awkward to express the guarantee. It is by employing realisability predicates that this awkwardness can be overcome. A realisability predicate exposes the underlying data in the index, so by taking proofs of realisability predicates *applied to the same index*, our *integrate* function gets precisely the guarantee that it needs. The ability to express the guarantee in this elegant manner is a demonstration of the strength of internalism. Thus the use of realisability predicates, which is central to externalist compositionality, can in fact be regarded as an application of an internalist technique to solve the compositionality problem of internalist datatypes.

Practically, how do we structure our libraries with the realisability transformation for better reusability? As McBride suggested, the datatypes should be delivered as codes and ornaments. The datatypes on which an operation is defined should be as general as possible, and other versions of the operation on more specialised types should be implemented in the form of realisability proofs. For example, *insert* should be defined for plain lists, and implemented for sorted lists and vectors as a function on sortedness proofs and length equalities respectively. Delivered in this way, then, *insert* for sorted lists, vectors, and sorted vectors can all be derived routinely by the realisability transformation as we have seen. This is the reusability and modularity offered by externalism. On the other hand, some operations are best defined on more specialised types, so preconditions can be cleanly expressed and manipulated. An example is the safe lookup function

$$\begin{array}{l} \mathit{lookup} : \{A : \mathsf{Set}\} \to \forall\ \{n\} \to \mathsf{Fin}\ n \to \mathsf{Vec}\ A\ n \to A \\ \mathit{lookup}\ \mathsf{fzero}\ \ (x :: xs) = x \\ \mathit{lookup}\ (\mathsf{fsuc}\ i)\ (x :: xs) = \mathit{lookup}\ i\ xs\ . \end{array}$$

It is natural to define this function on vectors (instead of lists) and use Fin (instead of Nat) as the index type, as the length constraint is embedded in the indices of the types of the data and requires no extra management, which is the advantage offered by internalism. So here is the development pattern we have in mind: Once a rich collection of ornaments are provided, programmers will have the freedom to choose which constraints they wish to impose on a basic type, compose the relevant ornaments and decode the composite ornament to a suitable inductive family T. Existing operations are upgraded to work with T routinely by the realisability transformation. And then, operations specific to T can be programmed directly on T, benefiting from the precision and convenience of programming with inductive families.

6. Related work

Section 2 is a faithful albeit condensed summary of McBride's original implementation of ornaments [11], except for a few notational changes. Our work is heavily based on algebraic ornaments and the associated construction method. Ornamental-algebraic ornaments have already appeared in McBride's original paper, and in particular, the *Length* predicate was derived from the ornament *VecO A*, which was one of our motivating examples. Also, a variant of less-than-or-equal-to relation on natural numbers was derived using an algebraic ornament by McBride, which led us to notice the similarity between Fin and _>_.

The idea of viewing vectors as realisability predicates was proposed by Bernardy [3, p 82], which refers to the realisability transformation defined for pure type systems by Bernardy and Lasson [4]. He started with the list type in which the element-type parameter is marked as "first-level," whereas the list type itself is "second-level." Applying the "projecting transformation," which removes first-level terms and demotes second-level terms to first-level, the second-level type of lists is transformed to the first-level type of natural numbers. And then, applying their realisability transformation, the list type is transformed to a second-level vector type indexed by first-level natural numbers. Our realisability interpretation can be seen as a translation of his idea into the language of ornaments without introducing levels: Our notion of complete objects and types would be second-level in Bernardy's system, while realisers and their types would be first-level. When applied to programs, their projecting transformation corresponds to our ornamental forgetful map. Due to the syntax-generic character of his transformations, Bernardy was able to derive vector append effortlessly from list append, and in particular deduce that, in the type of vector append, the index of the resulting vector is the sum of the indices of the two input vectors, because natural number addition is the (functional) realiser extracted from list append. Extraction of functional realisers from complete functions is not, and should not be, possible in our framework, however: The behaviour of a function taking a complete object may depend essentially on the added information, which is not available to a function taking only a realiser. For example, a function of type $\mathit{List\ Nat} \to \mathit{List\ Nat}$ may be defined to 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 $\mathit{Nat} \to \mathit{Nat}$ that reproduces the corresponding behaviour on natural numbers. On the other hand, it is reasonable to project list append to natural number addition, because list append is polymorphic and cannot inspect the elements. Indeed, in Bernardy and Lasson's system, it is impossible to produce second-level terms by induction on first-level terms, as the first-level terms are designed to be "computationally irrelevant" to second-level terms. This could be overcome by, for example, employing singleton types [12] to link different levels, but it can be inconvenient to do so explicitly. Our framework does not embody computational irrelevance, and trades the ability to derive polymorphic programs for simplicity and convenience.

The classic application of realisability in computing is program extraction, e.g., in Coq [15]. Terms are marked either as "informative" or "non-informative," and the non-informative terms, i.e., the proof terms irrelevant to computation, are removed during extraction, leaving the informative terms as the extracted program. It should be noted that our inverse transformation is not in general possible for other realisability systems, e.g., the one for the Calculus of Constructions in [15]. That is, it is not the case in general that having a realiser of a proposition implies that the proposition has a proof. Realisability in such systems can be used to show consistency of axioms — a proposition may not be provable, but can be postulated as an axiom consistently if it can be shown to be realisable. Our use of realisability terminology reflects that our development started from applying the notion to interpret ornamental-algebraic ornaments, but our development does not intend to follow faithfully those of the existing realisability theories and clearly deviates from those systems.

7. Future work

General ornament composition is a natural goal to pursue. A quick example that requires general ornament composition is *finite lists*, which are lists guaranteed to be shorter than a certain length:

$$\begin{array}{l} \textbf{data}\ \mathsf{FList}\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{fnil}\ \ : \forall\ \{m\} \to \mathsf{FList}\ (\mathsf{suc}\ m) \\ \quad \mathsf{fcons} : A \to \forall\ \{m\} \to \mathsf{FList}\ m \to \mathsf{FList}\ (\mathsf{suc}\ m)\ . \end{array}$$

The datatype comes out of composing the ornaments *ListO A* and *FinO*, neither of which is algebraic. One particular difficulty we encounter when trying to define general ornament composition is

that the new index set is a pullback, which is awkward to deal with. Also the implementation of *integrate* for general ornament composition is conceivably more complex. These should just be technical difficulties, though, and do not seem to detract from the feasibility of general ornament composition.

Before we commit ourselves to the implementation of general ornament composition, we may first consider increasing the expressive power of datatype descriptions and ornaments. For example, to define sorted lists without also indexing the type with a lower bound requires induction-induction [13]:

```
mutual
  data SList′ : Set where
    snil′  : SList′
    scons′ : (x : Nat) (xs : SList′) → x ≼ xs → SList′
  data _≼_ (y : Nat) : SList′ → Set where
    nil  : y ≼ snil′
    cons : ∀ {x xs b} → y ≤ x → y ≼ scons′ x xs b .
```

To talk about this and other similar datatypes, first we need to expand the universe to include codes for datatypes defined by induction-induction (or induction-recursion [7]). Another example is lists indexed with one of their prefixes:

```
data PList (A : Set) : List A → Set where
  pnil      : PList []
  pcons-[] : (x : A) → ∀ {xs} → PList xs → PList []
  pcons-:: : (x : A) → ∀ {xs} → PList xs → PList (x :: xs) .
```

It is possible to use the ornament

```
PListO : (A : Set) → Orn (List A) ! ⌊ListO A⌋
PListO A =
  σ Bool (false↦ say (ok []))
         true↦  σ A λx ↦ Δ (List A) λxs ↦
                  ask (ok xs) ∗
                    Δ Bool (false↦ say (ok [])
                            true↦  say (ok (x :: xs))))
```

which, in the cons case, inserts a boolean just before saying the index, which can be either [] or $x :: xs$, depending on the boolean. However, it is desirable to make the ornament reflect the fact that the native declaration has three constructors rather than two. To do so, we need to be able to refine the type Bool for the outermost σ to some three-element type. This requires expansion of the ornament language.

As with McBride's implementation of ornaments, we implement the realisability transformation in Agda just for experimenting with the idea, and do not intend to actually structure Agda programs with the combinators. To make the realisability transformation practically usable, it may have to be built into the language (along with ornaments) and supported by the development environment, allowing, e.g., automatic insertion of the transformation and inference of the datatype-generic parameters, or at least providing specific interactive commands to invoke the transformation, so the programmer need not bother with the details.

Theoretically, we may wish to get rid of the implementation details of datatype descriptions and ornaments, and examine all the concepts in terms of a cleaner mathematical semantics, like the one presented by Atkey, Johann, and Ghani [2]. Ornaments themselves now have an interesting compositional structure, so it is possible to develop an algebra of ornaments. Moreover, the correspondence between ornaments and realisability predicates looks like a subject ideally deserving a categorical treatment. We hope that our work will someday find a counterpart in the mathematical theory of datatypes, so it can be better characterised and understood.

Acknowledgements

We would like to thank Pierre-Évariste Dagand for referring us to Bernardy's idea, Shin-Cheng Mu for having the first discussion on this work with the first author, Liang-Ting Chen for suggesting the example of prefix-indexed lists, Jean-Philippe Bernardy and Fredrik Nordvall Forsberg for providing invaluable comments, and especially Conor McBride for sharing with us in the first place his unpublished work on ornaments. Meetings of the *Reusability and Dependent Types* project and *Algebra of Programming* research group greatly helped the development of our ideas. The first author is supported by the University of Oxford Clarendon Fund Scholarship, and both authors by the UK Engineering and Physical Sciences Research Council project *Reusability and Dependent Types*.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[2] R. Atkey, P. Johann, and N. Ghani. When is a type refinement an inductive type? In M. Hofmann, editor, *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 2011.

[3] J.-P. Bernardy. *A Theory of Parametric Polymorphism and an Application*. PhD thesis, Chalmers University of Technology, 2011.

[4] J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In M. Hofmann, editor, *Foundations of Software Science and Computation Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, 2011.

[5] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[6] J. Chapman, P.-É. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP '10, pages 3–14. ACM, 2010.

[7] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 1998.

[8] J. Gibbons. Fission for program comprehension. In T. Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, July 2006.

[9] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, December 1945.

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

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

[12] S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Programming Languages meets Program Verification*, PLPV '10, pages 1–8. ACM, 2010.

[13] F. Nordvall Forsberg and A. Setzer. Inductive-inductive definitions. In A. Dawar and H. Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer-Verlag, 2010.

[14] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming (AFP 2008)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

[15] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM, Jan. 1989.

[16] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.