Categorical Organisation of the Ornament–Refinement Framework

Hsiang-Shang Ko Jeremy Gibbons

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

Abstract

Dependently typed programming uses precise variants of data structures to ensure program correctness in an economical way, but designing reusable libraries for all possible variants of data structures is a difficult problem. The authors addressed the problem by extending McBride's *ornaments* to a framework of ornaments and *refinements* to support a modular structure for dependently typed libraries. We use lightweight category theory to organise the ornament–refinement framework and establish that *parallel composition* of ornaments, a key construction in the framework, gives rise to certain pullback properties. Two important sets of isomorphisms in the framework are then reconstructed using the pullback properties. This categorical organisation — which is completely formalised in the dependently typed language Agda — helps to separate the lower-level detail of the universes from the higher-level constructions of the isomorphisms, and to gain an abstract and effective understanding of the isomorphisms and the overall structure of the framework.

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

General Terms Design, Languages, Theory

Keywords Dependent types, category theory, modularity

1. Introduction

Dependent types provide a type discipline with which we can embed precise constraints in the types of data structures, thereby expressing more of our intentions in the type language, allowing the typechecker to verify more sophisticated correctness properties and even offer helpful type information during program development [14]. For example, instead of plain lists, depending on the context we can use vectors (lists indexed with their length), ordered lists, or ordered vectors to better express the properties a program satisfies and requires. Since elements of these datatypes encode correctness proofs in themselves, programs on these datatypes are correct by construction, avoiding the burden of providing separate proofs. This style, however, creates a serious problem for library construction: an ad hoc library would need to provide more or less the same set of operations for all possible variants of data structures and is hard to expand. For example, suppose that we have constructed a library for lists that include vectors, ordered lists, and ordered vectors, and now wish to add a new flavour of lists, say, association lists indexed by lists of keys. Operations need to be reimplemented not only for such key-indexed lists, but also for key-indexed vectors, ordered key-indexed lists, and ordered key-indexed vectors, even though key-indexing is the only new feature. A better structure for such a library would be having a separate module for each of the properties about length, ordering, and key-indexing. These modules can be developed separately, and there would be a way to assemble components in these modules at will, so, for example, ordered vectors and related operations would be synthesised from the components in the modules about length and ordering.

McBride [15] proposed *ornaments* as a mechanism for relating structurally similar datatypes (e.g., lists, vectors, and ordered lists), which was extended to a framework of ornaments and *refinements* in our previous work [10] to address the library structuring problem. Roughly speaking, an ornament from one datatype to another specifies how the second datatype is a more precise version of the first. For example, we have an ornament from lists to vectors which specifies how vectors are lists indexed by length information, and another ornament from lists to ordered lists which specifies what additional information ordered lists have with respect to lists. Compatible ornaments can be composed in parallel, giving rise to new datatypes and ornaments. For example, composing in parallel the ornament from lists to vectors and the ornament from lists to ordered lists produces the datatype of ordered vectors and an ornament from lists to ordered vectors. Ornaments induce refinements, which consist of isomorphisms that can be used to upgrade functions to more precise types. For example, an operation on vectors and ordered lists (e.g., the function that inserts an element into a list just before all larger elements, which increases the length of the list by one and preserves ordering), if expressed in some particular form specified by relevant refinements, can be combined into an operation on ordered vectors. Thus we get a way to fuse variants of datatypes and their operations into composite ones, which is exactly the modular structure we need for libraries.

In our previous work [10], we gave a purely type-theoretical presentation of the ornament–refinement framework in the dependently typed language Agda [3, 17, 18], and constructed several key isomorphisms in the framework by datatype-generic induction. Although there are intuitions that explain why these isomorphisms exist, these high-level intuitions are not manifested in the low-level constructions of the isomorphisms. Type theory provides a precise but basic mathematical language that has particular support for inductive definitions; for more complicated constructions, however, it is better to employ more abstract theories to properly express the ideas. In the case of the ornament–refinement framework, a *categorical* organisation would be helpful: categorical notions that more closely correspond to the intuitions would emerge from the type-theoretic constructions, and these categorical notions would form a foundation for alternative, higher-level constructions of the key isomorphisms, providing more insight into their existence.

We present in this paper a categorical organisation of the ornament–refinement framework (a type-theoretic summary of the framework is given in Section 2), with the primary aim of providing new constructions of the key isomorphisms. In more detail:

- A small part of category theory is formalised in Agda (Section 3), so the type-theoretical constructions in the framework and the categorical abstractions can be seamlessly presented in one uniform language.
- Various constructions in the framework are organised under certain categories and functors (Sections 4.1 and 4.2), giving a clearer view of the overall structure of the framework.
- On top of the categorical organisation, we show that parallel composition of ornaments gives rise to certain *pullback* properties (Section 4.3).
- These pullback properties then play essential roles in our new constructions of the key isomorphisms in the framework (Section 5).

At the end, we discuss what we have gained from the categorical organisation, and compare our work to the categorical treatment of ornaments offered by Dagand and McBride [7], concluding with a remark on our style of presentation (Section 6).

2. The ornament–refinement framework

We begin with a condensed overview of the ornament–refinement framework [10]. At the core of the framework are universe constructions [1, 11] for *index-first datatypes* [5] and their ornamental relationships. (We use "datatype" as a synonym for "inductive family" in this paper.) Rather than going into the detail of the universe constructions, here we postulate the existence of the universes and only present examples using high-level datatype declarations; these datatype declarations can be elaborated into codes in the universes by some mechanism like the one given by Dagand and McBride [8].

2.1 Index-first datatypes and the universe of datatype descriptions

In Agda, an inductive family is declared by listing all possible constructors and their types, all ending with one of the types in that inductive family. For index-first datatypes, however, the logical order is just the opposite: we list all possible patterns of (the indices of) the types in the inductive family, and specify for each pattern which constructors it accepts. Because of this important change of logical order, Dagand and McBride [6] proposed a new notation for index-first datatypes, which was later adapted in our previous work [10] to be slightly more Agda-like. In our adaptation of the notation, an index-first datatype declaration is explicitly prefixed by the keyword **indexfirst**. Index-first versions of simple datatypes look almost like Haskell data declarations. For example, natural numbers are declared by

```
indexfirst data Nat : Set where
  Nat ∋ nil
      | cons (n : Nat)
```

The only possible pattern of the datatype is Nat, which accepts two constructors nil and cons, the latter taking a recursive argument named n. (We name the constructors nil and cons instead of the more conventional zero and suc in order to make explicit later the connections to list-like datatypes.) We declare lists similarly: Let A : Set, which we will directly refer to in subsequent code as if it were a module parameter. The datatype of lists is declared as

```
indexfirst data List A : Set where
  List A ∋ nil
         | cons (x : A) (xs : List A)
```

The declaration of vectors, i.e., lists indexed with their length, is more interesting, fully exploiting the power of index-first datatypes.

```
indexfirst data Vec A : Nat → Set where
  Vec A nil        ∋ nil
  Vec A (cons n) ∋ cons (x : A) (xs : Vec A n)
```

Vec A is a family of types indexed by Nat, and we do pattern matching on the index, splitting the datatype into two cases Vec A nil and Vec A (cons n) for some n : Nat. The first case only accepts the nil constructor, and the second case only accepts the cons constructor. Because the form of the index restricts constructor choice, the structure of a vector xs : Vec A n must follow that of n, i.e., the number of cons nodes in xs must match that in n. Also note that the index-first concept enables optimisation of the representation of vectors — no information needs to be stored in a cons node other than the head and the tail, so the representation of a vector is as efficient as that of an ordinary list [4].

The ornament–refinement framework has a parametrised universe for index-first datatypes:

$$\mathsf{Desc} : \mathsf{Set} \to \mathsf{Set}_1$$

The universe is an inductive family itself; we omit the details from this paper. Elements in the universe are called (datatype) *descriptions*. A description of type Desc I for some I : Set is an encoding of a datatype indexed by I. That is, there is a least fixed-point operator

$$\mu : \{I : \mathsf{Set}\} \to \mathsf{Desc}\, I \to (I \to \mathsf{Set})$$

which decodes descriptions into actual datatypes. The declarations of Nat, List A, and Vec A are merely more readable presentations of datatype descriptions:

- natural numbers are considered to be a type family trivially indexed by the one-element set $\top$ (whose only element is tt), so the type Nat is actually μ *NatD* tt for some description *NatD* : Desc $\top$;
- lists are also trivially indexed, but have a Set parameter, so List A is actually μ (*ListD* A) tt where *ListD* : Set $\to$ Desc $\top$ is a family of descriptions indexed by Set;
- vectors are indexed by Nat and have a Set parameter, so Vec A n is actually μ (*VecD* A) n for some *VecD* : Set $\to$ Desc Nat.

Accompanying the universe are generic fold and induction operators parametrised by descriptions, so it is possible to write datatype-generic inductive programs and proofs once and for all for all datatypes encoded by the universe at once.

2.2 Ornaments

The three datatypes Nat, List A, and Vec A are obviously structurally similar: a list is a natural number whose cons nodes store elements of A, and a vector is a list whose type is enriched with length information. Such a relationship between structurally similar datatypes is captured by *ornaments*, which encode differences between datatype declarations. For example,

- compared to the declaration of Nat, the declaration of List A has an extra field (x : A) in the cons constructor;
- compared to the declaration of List A, the declaration of Vec A (i) has index set Nat rather than $\top$, (ii) lacks the cons constructor in the nil case and the nil constructor in the cons case, and (iii) has a more refined index n rather than tt at the recursive position.

An important property of any ornament is that it necessarily relates a basic datatype to a more informative variant, so from the ornament we can derive a forgetful function from the more informative datatype to the basic one, erasing information according to the ornament's specification of datatype differences. For example, there is a forgetful function from lists to natural numbers that discards elements stored in cons nodes, so it computes the length of a list, and there is another forgetful function from vectors to lists which returns the list underlying a vector.

Ornaments constitute the second universe used in the ornament–refinement framework:

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

An ornament $O : \mathsf{Orn}\ e\ D\ E$ from the basic description D to the more informative description E is parametrised by a function e from the index set of E to that of D, and encodes structural differences between D and E. From the ornament O we can derive a forgetful function of type

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

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

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

For example, there are families of ornaments

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

and

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

(where $! = \lambda\_ \mapsto \mathsf{tt}$) that encode the structural differences between natural numbers, lists, and vectors described above. The function

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

computes the length of a list, and the function

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

computes the list underlying a vector.

2.3 Ornamental descriptions

It is common to modify an existing basic description into a more informative new description such that there is an ornament between the two descriptions. For example, we might use the declaration of natural numbers as a template and add a field to the cons constructor to get the declaration of lists. The ornament–refinement framework has a third universe of *ornamental descriptions* that lets us write a new description relative to a basic description such that there is an ornament from the basic description to the new one:

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

An ornamental description

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

might be thought of as simultaneously denoting both a new description of type $\mathsf{Desc}\ J$ and an ornament from the basic description D to that new description. We use floor and ceiling brackets $\lfloor\_\rfloor$ and $\lceil\_\rceil$ to resolve ambiguity: the new description is

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

and the ornament is

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

For example, consider the following *singleton datatype*:

```
indexfirst data ListS A : List A → Set where
  ListS nil         ∋ nil
  ListS (cons x xs) ∋ cons (s : ListS xs)
```

For each type $\mathsf{ListS}\ A\ \mathit{xs}$, there is exactly one (canonical) element, which has the same structure (i.e., number of cons nodes) as xs. There is an ornament from $\mathsf{List}\ A$ to $\mathsf{ListS}\ A$ which does pattern matching on the index, in each case restricts the constructor choice to the one matched against, and in the cons case deletes the element and sets the index of the recursive position to be the value of the tail in the pattern. In general, for any description $D : \mathsf{Desc}\ I$, there is an ornamental description

$$\mathit{singOrn}\ D : \mathsf{OrnDesc}\ (\Sigma\ I\ (\mu\ D))\ \mathsf{proj}_1\ D$$

called the *singleton ornament* on D, which delivers a singleton datatype and an ornament from D to that datatype, such that

$$\mathit{forget}\ \lceil \mathit{singOrn}\ D \rceil\ \{i,x\}\ s \equiv x$$

for all $i : I$, $x : \mu\ D\ i$, and $s : \mu\ \lfloor \mathit{singOrn}\ D \rfloor\ (i,x)$ (where $\_\!\equiv\!\_$ denotes propositional equality). For example, $\mathsf{ListS}\ A\ \mathit{xs}$ desugars to $\mu\ \lfloor \mathit{singOrn}\ (\mathit{ListD}\ A) \rfloor\ (\mathsf{tt},\mathit{xs})$, and the forgetful function

$$\begin{aligned}&\lambda\{\mathit{xs}\} \mapsto \mathit{forget}\ \lceil \mathit{singOrn}\ (\mathit{ListD}\ A) \rceil\ \{\mathsf{tt},\mathit{xs}\}\\ &\qquad\qquad : \forall\ \{\mathit{xs}\} \to \mathsf{ListS}\ A\ \mathit{xs} \to \mathsf{List}\ A\end{aligned}$$

can be proved to satisfy

$$\mathit{forget}\ \lceil \mathit{singOrn}\ (\mathit{ListD}\ A) \rceil\ \{\mathsf{tt},\mathit{xs}\}\ s \equiv \mathit{xs}$$

for all $\mathit{xs} : \mathsf{List}\ A$ and $s : \mathsf{ListS}\ A\ \mathit{xs}$. Specifically for singleton datatypes, there is a function that returns the sole element of any singleton type:

$$\begin{aligned}\mathit{singleton} :\ &\{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\\ &\{i : I\}\ (x : \mu\ D\ i) \to \mu\ \lfloor \mathit{singOrn}\ D \rfloor\ (i,x)\end{aligned}$$

We will shortly see that singleton datatypes play a key role in the framework.

```
data _⁻¹_ {I J : Set} (e : J → I) : I → Set where
  ok : (j : J) → e ⁻¹ (e j)

und : {I J : Set} {e : J → I} {i : I} → e ⁻¹ i → J
und (ok j) = j

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

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

π₁ : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → J
π₁ = und ∘ _⋈_.j

π₂ : {I J K : Set} {e : J → I} {f : K → I} → e ⋈ f → K
π₂ = und ∘ _⋈_.k
```

Figure 1. Definitions of inverse images and set-theoretic pullbacks.

2.4 Composition of ornaments

The ornament–refinement framework offers two ways to compose ornaments, *sequential composition* and *parallel composition*. Let $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$, and $F : \mathsf{Desc}\ K$. Sequential composition collapses an ornament from D to E and another ornament from E to F into one directly from D to F:

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

Parallel composition is more interesting. Given two ornaments $O : \mathsf{Orn}\ e\ D\ E$ and $P : \mathsf{Orn}\ f\ D\ F$ with a common basic description, parallel composition of O and P produces a new description that is a variant of D, i.e., it is an ornamental description relative to D:

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

The index set $e \bowtie f$ is the set-theoretic pullback of $e : J \to I$ and $f : K \to I$, and $\mathit{pull} : e \bowtie f \to I$ is the usual projection of pullbacks.

Their definitions are shown in Figure 1. Here we offer an intuitive explanation of what parallel composition does: since both O and P encode modifications to the same basic description D, we can commit all the modifications to D to get a new description $\lfloor O \otimes P \rfloor$, and merge all the modifications into one ornament $\lceil O \otimes P \rceil$. For example, let $\_\leqslant_A\_ : A \to A \to \mathsf{Set}$ be an ordering on A and declare a datatype of ordered lists indexed by a lower bound under this ordering:

```
indexfirst data OrdList A _⩽A_ : A → Set where
  OrdList A _⩽A_ a
    ∋ nil
    | cons (x : A) (leq : a ⩽A x) (xs : OrdList A _⩽A_ x)
```

We use an ornamental description to express that $\mathsf{OrdList}\ A\ \_\leqslant_A\_$ is a variant of $\mathsf{List}\ A$:

```
OrdListOD A _⩽A_ : OrnDesc A ! (ListD A)
```

which inserts the field *leq* and refines the indices. Parallel composition of $\lceil \mathit{OrdListOD}\ A\ \_\leqslant_A\_ \rceil$ and *ListD-VecD A* then produces (i) a new datatype of ordered vectors

```
indexfirst data OrdVec A _⩽A_ : A → Nat → Set where
  OrdVec A _⩽A_ a nil        ∋ nil
  OrdVec A _⩽A_ a (cons n)
    ∋ cons (x : A) (leq : a ⩽A x) (xs : OrdVec A _⩽A_ x n)
```

and (ii) an ornament from which we can derive a forgetful function from ordered vectors to plain lists that retains only the elements. It is conceivable, though, that there should also be ornaments that give rise to "less forgetful" functions, converting an ordered vector to an ordered list or a vector. Indeed, for any parallel composition $O \otimes P$ there are two *difference ornaments*

```
diffOrn-l O P : Orn π₁ E ⌊O ⊗ P⌋
diffOrn-r O P : Orn π₂ F ⌊O ⊗ P⌋
```

such that both

```
forget O ∘ forget (diffOrn-l O P)
```

and

```
forget P ∘ forget (diffOrn-r O P)
```

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

2.5 Optimised predicates and ornamental promotion isomorphisms

Let $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$, and $O : \mathsf{Orn}\ e\ D\ E$. An important construction for the ornament O is the parallel composition of O and the singleton ornament on D. To see why it is important, consider the ornament $\lceil \mathit{OrdListOD}\ A\ \_\leqslant_A\_ \rceil$ from lists to ordered lists given above. Composing it with the singleton ornament on *ListD A* in parallel results in the following datatype:

```
indexfirst data Ordered A _⩽A_ : A → List A → Set where
  Ordered A _⩽A_ a nil            ∋ nil
  Ordered A _⩽A_ a (cons x xs)
    ∋ cons (leq : a ⩽A x) (s : Ordered A _⩽A_ x xs)
```

A proof of $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a\ \mathit{xs}$ consists of a series of inequality proofs which ensures that *xs* is ordered and bounded below by a, so $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a$ is a predicate which characterises lists that are ordered and bounded. The Ordered predicate is useful because of the following family of isomorphisms

$$\mathsf{OrdList}\ A\ \_\leqslant_A\_\ a\ \cong\ \Sigma[\mathit{xs} : \mathsf{List}\ A]\ \mathsf{Ordered}\ A\ \_\leqslant_A\_\ a\ \mathit{xs}$$

for all $a : A$ — an ordered list bounded below by a can be converted to/from a plain list and a proof that it is ordered and bounded below by a. In general, define the *optimised predicates* for the ornament O as the parallel composition of O and the singleton ornament on D:

```
OptP O : {i : I} → e ⁻¹ i → μ D i → Set
OptP O {i} j x = μ ⌊O ⊗ ⌈singOrn D⌉⌋ (j, (ok (i,x)))
```

(so $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a$ desugars to $\mathit{OptP}\ \lceil \mathit{SListOD}\ A\ \_\leqslant_A\_ \rceil\ (\mathsf{ok}\ a)$) then we have the (ornamental) *promotion isomorphisms*

```
ornPromIso O :
  {i : I} (j : e ⁻¹ i) → μ E (und j) ≅ Σ[x : μ D i] OptP O j x
```

such that $\mathsf{proj}_1 \circ \mathsf{Iso}.\mathit{to}\ (\mathit{ornPromIso}\ O\ (\mathsf{ok}\ j'))$ for any $j' : J$ is definitionally $\mathit{forget}\ O\ \{j'\}$. (For an isomorphism $\mathit{iso} : X \cong Y$, we denote its left-to-right direction as $\mathsf{Iso}.\mathit{to}\ \mathit{iso} : X \to Y$ and its right-to-left direction as $\mathsf{Iso}.\mathit{from}\ \mathit{iso} : Y \to X$. The full definition of the record Iso will be given in Section 3.) The promotion isomorphisms are named as such because the right-to-left direction can be interpreted as promoting an element of the basic type (e.g., a list) to an element of the more informative type (e.g., an ordered list) provided that there is a proof that the optimised predicate (e.g., Ordered) is satisfied.

2.6 Refinements

The ornament–refinement framework recognises the promotion isomorphisms as a key abstraction, and axiomatises the existence of promotion isomorphisms between two types as *refinements*:

```
record Refinement (X Y : Set) : Set₁ where
  field
    P : X → Set
    i : Y ≅ Σ X P
  forget : Y → X
  forget = proj₁ ∘ Iso.to i
```

A type X is refined by a type Y if there is a *promotion predicate* P on X and a promotion isomorphism from X to Y that uses P. In a refinement $r : \mathsf{Refinement}\ X\ Y$, we denote the part of the promotion isomorphism $\mathsf{proj}_1 \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.\mathit{i}\ r) : Y \to X$ by $\mathsf{Refinement}.\mathit{forget}\ r$, since it is an abstraction of the forgetful function derived from an ornament, demoting elements of the more refined type Y to the basic type X. For example, for any $a : A$, there is a refinement from $\mathsf{List}\ A$ to $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ a$ whose promotion predicate is $\mathsf{Ordered}\ A\ \_\leqslant_A\_\ a$ and whose promotion isomorphism is the one derived from the ornament from lists to ordered lists:

```
List-OrdListR :
  (a : A) → Refinement (List A) (OrdList A _⩽A_ a)
List-OrdListR a = record
  { P = Ordered A _⩽A_ a
  ; i = ornPromIso ⌈OrdListOD A _⩽A_⌉ (ok a)}
```

and $\mathsf{Refinement}.\mathit{forget}\ (\mathit{List\text{-}OrdListR}\ a)$ is the ornamental forgetful function from $\mathsf{OrdList}\ A\ \_\leqslant_A\_\ a$ to $\mathsf{List}\ A$. Refinements are a key abstraction because they provide basic building blocks for upgrading functions to more informative types. For example, given $a : A$, to promote a function

```
f : List A → List A
```

to the more informative type

```
OrdList A _⩽A_ a → OrdList A _⩽A_ a
```

it suffices to provide a proof of type

```
∀ xs → Ordered A _⩽A_ a xs → Ordered A _⩽A_ a (f xs)
```

i.e., a proof that f preserves ordering (and the lower bound a). This proof can then be combined with f into a function on ordered lists using the promotion isomorphism from lists to ordered lists; the procedure is captured in the following function:

```
upgrade :
  {X Y Z W : Set}
  (r : Refinement X Y) (s : Refinement Z W)
  (f : X → Z) →
  (∀ x → Refinement.P r x → Refinement.P s (f x)) →
  Y → W
```

$$\begin{array}{l} \mathit{upgrade}\ r\ s\ f\ p\ = \\ \quad \mathsf{Iso}.\mathit{from}\ (\mathsf{Refinement}.\mathit{i}\ s) \circ (f * p) \circ \mathsf{Iso}.\mathit{to}\ (\mathsf{Refinement}.\mathit{i}\ r) \end{array}$$

where the operator $\_{*}\_$ is defined by $g * h = \lambda (x,y) \mapsto (g\,x, h\,y)$. Following this path, we could reach a refinement-based function upgrading mechanism (beyond the scope of this paper) that generalises Dagand and McBride's "functional ornaments" [6]: the refinement-based mechanism would be more flexible because refinements allow us to separate two constructions — (i) ornamental relationships between inductive families, from which we can derive particular promotion isomorphisms between corresponding types in the inductive families, and (ii) how promotion isomorphisms in general enable function upgrading — whereas functional ornaments unnecessarily couple these two constructions.

An ornament gives rise to a family of promotion isomorphisms — and thus refinements — between corresponding types in two inductive families. More specifically: if $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$, and $O : \mathsf{Orn}\ e\ D\ E$, then the refinements induced by O are between $\mu\ D\ i$ and $\mu\ E\ (\mathit{und}\ j)$ for all $i : I$ and $j : e^{-1}\ i$. We thus postulate *refinement families* between two type families as

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

Now we can summarise the roles of the optimised predicates and the ornamental promotion isomorphisms in one construction:

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

That is, every ornament induces a refinement family that uses the optimised predicates and the ornamental promotion isomorphisms. This is called the *refinement semantics of ornaments* in our previous work [10].

2.7 Predicate swapping and the modularity isomorphisms

Between two types, we can have multiple refinements that use different promotion predicates. Moreover, we can replace the promotion predicate of a refinement with a new predicate, resulting in a new refinement between the same types that uses the new predicate, provided that we can supply a proof that the original predicate and the new one are pointwise isomorphic — so a promotion isomorphism can be derived for the new refinement. Formally, we have

$$\begin{array}{l} \mathit{swapR} : \\ \quad \{X\ Y : \mathsf{Set}\}\ (r : \mathsf{Refinement}\ X\ Y) \\ \quad (Q : X \to \mathsf{Set})\ (\mathit{isos} : \forall\, x \to \mathsf{Refinement}.P\ r\ x \cong Q\ x) \to \\ \quad \mathsf{Refinement}\ X\ Y \end{array}$$

such that the promotion predicate of a refinement $\mathit{swapR}\ r\ Q\ \mathit{isos}$ is definitionally Q. This is called *predicate swapping* and can be easily extended to work for refinement families as well: we have

$$\begin{array}{l} \mathit{swapFR} : \\ \quad \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ \{e : J \to I\} \\ \quad (\mathit{rs} : \mathit{FRefinement}\ e\ X\ Y) \\ \quad (Q : \{i : I\}\ (j : e^{-1}\ i) \to X\ i \to \mathsf{Set}) \\ \quad (\mathit{isos} : \{i : I\}\ (j : e^{-1}\ i) \\ \qquad\qquad (x : X\ i) \to \mathsf{Refinement}.P\ (\mathit{rs}\ j)\ x \cong Q\ j\ x) \to \\ \quad \mathit{FRefinement}\ e\ X\ Y \end{array}$$

such that $\mathsf{Refinement}.P\ (\mathit{swapFR}\ \mathit{rs}\ Q\ \mathit{isos}\ j)$ is definitionally $Q\ j$ for all $i : I$ and $j : e^{-1}\ i$.

One situation where predicate swapping is helpful is when we try to use the refinement semantics of parallel composition. For example, consider the datatype $\mathsf{OrdVec}\ A\ \_{\leqslant_A}\_$ of ordered vectors. The refinement semantics of the parallel composition underlying $\mathsf{OrdVec}\ A\ \_{\leqslant_A}\_$ uses the following monolithic datatype for its promotion predicates:

$$\begin{array}{llll} \multicolumn{4}{l}{\textbf{data}\ \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_ : A \to \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}} \\ \multicolumn{4}{l}{\quad\textbf{where}} \\ \quad \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ \mathsf{nil} & \mathsf{nil} & & \ni \mathsf{nil} \\ \quad \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ (\mathsf{cons}\ n) & \mathsf{nil} & & \not\ni \\ \quad \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ \mathsf{nil} & (\mathsf{cons}\ x\ \mathit{xs}) & & \not\ni \\ \quad \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ (\mathsf{cons}\ n) & (\mathsf{cons}\ x\ \mathit{xs}) & & \\ \multicolumn{4}{l}{\qquad \ni \mathsf{cons}\ (\mathit{leq} : a \leqslant_A x)\ (p : \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ n\ \mathit{xs})} \end{array}$$

(A '$\not\ni$' symbol following a case of a datatype means that the case does not accept any construtor, i.e., the case is uninhabited.) However, if we are constructing a minimal library of list-like datatypes and operations on them, $\mathsf{OrderedLength}$ should not have a place in this library, since it can be modularly composed from simpler datatypes: there are isomorphisms

$$\begin{array}{l} \mathsf{OrderedLength}\ A\ \_{\leqslant_A}\_\ a\ n\ \mathit{xs} \\ \quad \cong\ \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ a\ \mathit{xs} \times \mathsf{Length}\ A\ n\ \mathit{xs} \end{array} \tag{1}$$

for all $a : A$, $n : \mathsf{Nat}$, and $\mathit{xs} : \mathsf{List}\ A$, where $\mathsf{Length}\ A$ is the optimised predicate datatype derived from the ornament from lists to vectors:

$$\begin{array}{llll} \multicolumn{4}{l}{\textbf{data}\ \mathsf{Length}\ A : \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where}} \\ \quad \mathsf{Length}\ A\ \mathsf{nil} & \mathsf{nil} & & \ni \mathsf{nil} \\ \quad \mathsf{Length}\ A\ (\mathsf{cons}\ n) & \mathsf{nil} & & \not\ni \\ \quad \mathsf{Length}\ A\ \mathsf{nil} & (\mathsf{cons}\ x\ \mathit{xs}) & & \not\ni \\ \quad \mathsf{Length}\ A\ (\mathsf{cons}\ n) & (\mathsf{cons}\ x\ \mathit{xs}) & & \ni \mathsf{cons}\ (l : \mathsf{Length}\ A\ n\ \mathit{xs}) \end{array}$$

The library would provide proofs of properties about operations on Lists expressed in terms of Ordered and Length, and these operations can be upgraded for OrdVec by the previously sketched function upgrading mechanism. For example, suppose we have in the library a list operation

$$f : \mathsf{List}\ A \to \mathsf{List}\ A$$

and two proofs that this operation preserves ordering and length:

$$\begin{array}{l} p : \forall\, a\ \mathit{xs} \to \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ a\ \mathit{xs} \to \\ \qquad\qquad\quad\ \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ a\ (f\ \mathit{xs}) \\ q : \forall\, n\ \mathit{xs} \to \mathsf{Length}\ A\ n\ \mathit{xs} \to \mathsf{Length}\ A\ n\ (f\ \mathit{xs}) \end{array}$$

then we can upgrade f to an operation on ordered vectors as follows:

$$\begin{array}{l} f' : \forall\, \{a\ n\} \to \mathsf{OrdVec}\ A\ \_{\leqslant_A}\_\ a\ n \to \mathsf{OrdVec}\ A\ \_{\leqslant_A}\_\ a\ n \\ f'\ \{a\}\ \{n\}\ =\ \textbf{let}\ r : \mathsf{Refinement}\ (\mathsf{List}\ A)\ (\mathsf{OrdVec}\ A\ \_{\leqslant_A}\_\ a\ n) \\ \qquad\qquad\qquad\quad\ \ r\ =\ \{\ \}_0 \\ \qquad\qquad\quad\ \ \textbf{in}\ \mathit{upgrade}\ r\ r\ f\ (\lambda \mathit{xs} \mapsto p\ a\ \mathit{xs} * q\ n\ \mathit{xs}) \end{array}$$

where r is the refinement obtained by predicate swapping on the refinement semantics of OrdVec with the isomorphisms (1), so the promotion predicate of r is $\lambda \mathit{xs} \mapsto \mathsf{Ordered}\ A\ \_{\leqslant_A}\_\ a\ \mathit{xs} \times \mathsf{Length}\ A\ n\ \mathit{xs}$. In general, if there are ornaments $O : \mathsf{Orn}\ e\ D\ E$ and $P : \mathsf{Orn}\ f\ D\ F$ (which can be composed in parallel) where $D : \mathsf{Desc}\ I$, $E : \mathsf{Desc}\ J$, and $F : \mathsf{Desc}\ K$, then we have the *modularity isomorphisms*

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

for all $i : I$, $j : e^{-1}\ i$, $k : f^{-1}\ i$, and $x : \mu\ D\ i$. That is, the optimised predicate for $\lceil O \otimes P \rceil$ is pointwise isomorphic to (and thus can be swapped with) the pointwise conjunction of the optimised predicates for O and P. The isomorphisms (1) are just an instance of the modularity isomorphisms.

The key constructions in the ornament–refinement framework are the ornamental promotion isomorphisms and the modularity isomorphisms, the former leading to the refinement semantics of ornaments (which provides basic building blocks for function upgrading) and the latter making modular library structure possible with parallel composition. In our previous paper [10], the two sets of isomorphisms are constructed in an ad hoc manner, but the implementations have some deep similarities and call for unification. Indeed, a more unified treatment is possible: we will show in Section 5 that both sets of isomorphisms stem from categorical pullback properties derived from parallel composition (to be stated in Section 4.3).

3. Formalisation of categories

In this section we formalise some basic category-theoretic terms in Agda, establishing vocabulary for Sections 4 and 5.

3.1 Definitions of categories and functors

We will define a category to be a set of objects and sets of morphisms indexed by source and target, together with the usual laws. Special attention must be paid to equality on morphisms, though, which is usually coarser than definitional equality — for example, in the category of sets and (total) functions, it is necessary to identify functions up to extensional equality, so uniqueness of morphisms in universal properties would make sense. One simple way to achieve this in Agda's intensional setting is to use *setoids* [2] — i.e., sets with an explicitly specified equivalence relation — to represent sets of morphisms. Subsequently, all operations on morphisms should respect the equivalence.

In Agda, the type of setoids can be defined as a record which contains a carrier set, an equivalence relation on the set, and the three laws for the equivalence relation:[1]

```
record Setoid {c d : Level} : Set (suc (c ⊔ d)) where
  field
    Carrier : Set c
    _≈_     : Carrier → Carrier → Set d
    refl    : ∀ {x} → x ≈ x
    sym     : ∀ {x y} → x ≈ y → y ≈ x
    trans   : ∀ {x y z} → x ≈ y → y ≈ z → x ≈ z
```

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

```
FunSetoid : Set → Set → Setoid
FunSetoid A B = record { Carrier = A → B
                       ; _≈_     = _≐_
                       ; proofs of laws }
```

where $\_\doteq\_$ is defined by $f \doteq g = \forall x \to f\ x \equiv g\ x$. Proofs of the three laws are omitted from the paper.

Similarly, we can define the type of categories as a record containing a set of objects, a collection of setoids of morphisms indexed by source and target (so morphisms with the same source and target — and only such morphisms — can be compared for equality), the composition operator on morphisms, the identity morphisms, and the laws of categories. The definition is shown in Figure 2. Two notations are introduced to improve readability: $X \Rightarrow Y$ is defined to be the carrier set of the setoid of morphisms from X to Y, and $f \approx g$ is defined to be the equivalence between the morphisms f and g as specified by the setoid to which f and g belong. The last two laws *cong-l* and *cong-r* require composition of morphisms to respect the equivalence on morphisms; they are given in this form to work better with the equational reasoning combinators commonly used in Agda (see, for example, the AoPA library [16]). Now we can define the category 𝔽un of sets and (total) functions as

```
𝔽un : Category
𝔽un = record { Object    = Set
             ; Morphism  = FunSetoid
             ; _·_ = _∘_
             ; id  = λx ↦ x
             ; proofs of laws }
```

```
record Category {l m n : Level} : Set (suc (l ⊔ m ⊔ n)) where
  field
    Object   : Set l
    Morphism : Object → Object → Setoid {m} {n}
  _⇒_ : Object → Object → Set m
  X ⇒ Y = Setoid.Carrier (Morphism X Y)
  _≈_ : ∀ {X Y} → X ⇒ Y → X ⇒ Y → Set n
  _≈_ {X} {Y} = Setoid._≈_ (Morphism X Y)
  field
    _·_ : ∀ {X Y Z} → Y ⇒ Z → X ⇒ Y → X ⇒ Z
    id  : ∀ {X} → X ⇒ X
    id-l   : ∀ {X Y} (f : X ⇒ Y) → id · f ≈ f
    id-r   : ∀ {X Y} (f : X ⇒ Y) → f · id ≈ f
    assoc  : ∀ {X Y Z W}
             (f : Z ⇒ W) (g : Y ⇒ Z) (h : X ⇒ Y) →
             (f · g) · h ≈ f · (g · h)
    cong-l : ∀ {X Y Z} {f g : Y ⇒ Z} (h : X ⇒ Y) →
             f ≈ g → f · h ≈ g · h
    cong-r : ∀ {X Y Z} (h : Y ⇒ Z) {f g : X ⇒ Y} →
             f ≈ g → h · f ≈ h · g
```

Figure 2. Definition of categories.

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

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

Note that the equivalence on morphisms is defined to be componentwise extensional equality, which is formulated with the help of McBride's "John Major" heterogeneous equality $\_\simeq\_$ [13] — the equivalence $\_\dot{\simeq}\_$ is defined by $g \mathrel{\dot{\simeq}} h = \forall x \to g\ x \simeq h\ x$. (Given $y : Y\ j$ for some $j : J$, the types of $u\ \{j\}\ y$ and $u'\ \{j\}\ y$ are not definitionally equal but only provably equal, so it is necessary to employ heterogeneous equality.)

We will also need functors, whose definition is shown in Figure 3: a functor consists of two mappings, one on objects and the other on morphisms, where the morphism part respects equivalence on morphisms and preserves identity and composition. For exam-

[1] The definition of setoids uses Agda's universe polymorphism, so the definition can be instantiated at suitable levels of the Set hierarchy as needed. We will give the first few universe-polymorphic definitions with full detail about the levels, but will later switch to writing just 'Set _' to suppress the noise.

```
record Functor
  {l m n l′ m′ n′ : Level}
  (C : Category {l} {m} {n}) (D : Category {l′} {m′} {n′}) :
  Set (l ⊔ m ⊔ n ⊔ l′ ⊔ m′ ⊔ n′) where
  field
    object    : Object_C → Object_D
    morphism : ∀ {X Y} → X ⇒_C Y → object X ⇒_D object Y
    ≈-respecting :
      ∀ {X Y} {f g : X ⇒_C Y} →
      f ≈_C g → morphism f ≈_D morphism g
    id-preserving :
      ∀ {X} → morphism (id_C {X}) ≈_D id_D {object X}
    comp-preserving :
      ∀ {X Y Z} (f : Y ⇒_C Z) (g : X ⇒_C Y) →
      morphism (f ·_C g) ≈_D (morphism f ·_D morphism g)
```

Figure 3. Definition of functors.

ple, we have two forgetful functors from $\mathbb{F}$AM to $\mathbb{F}$UN, one summing components together

```
FAMF : Functor 𝔽AM 𝔽UN
FAMF = record { object    = λ(I,X) ↦ Σ I X
              ; morphism = λ(e,u) ↦ e ∗ u
              ; proofs of laws }
```

and the other extracting the index part.

```
FAMI : Functor 𝔽AM 𝔽UN
FAMI = record { object    = λ(I,X) ↦ I
              ; morphism = λ(e,u) ↦ e
              ; proofs of laws }
```

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

$$(e \doteq f) \times (\forall \{j\} \to u\,\{j\} \mathrel{\dot{\simeq}} v\,\{j\}) \to (e * u \doteq f * v)$$

and for FAMI we need to prove

$$(e \doteq f) \times (\forall \{j\} \to u\,\{j\} \mathrel{\dot{\simeq}} v\,\{j\}) \to (e \doteq f)$$

both of which can be easily discharged.

3.2 Definition of pullbacks

We will define a pullback to be a product in the suitable slice category, where a product is defined to be a terminal object in the suitable span category. Below we give definitions of all these terms in logical order. Let C : Category in what follows.

- A *slice category* based on C is parametrised by an object B; its objects are those morphisms in C with target B and its morphisms are mediating morphisms giving rise to commutative triangles — diagrammatically,

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

  The slice objects and morphisms are defined in Agda as two records; they are shown in the upper half of Figure 4 along with the definition of slice categories. Note that the equivalence on slice morphisms is defined as only the equivalence on the mediating morphisms, essentially achieving proof-irrelevance.

```
record Slice C B : Set _ where
  constructor slice
  field
    T  : Object
    s  : T ⇒ B
record SliceMorphism C B (s t : Slice C B) : Set _ where
  constructor sliceMorphism
  field
    m : Slice.T s ⇒ Slice.T t
    triangle : Slice.s t · m ≈ Slice.s s
SliceCategory C B : Category
SliceCategory C B =
  record
    { Object     = Slice C B
    ; Morphism =
        λs t ↦ record
                  { Carrier = SliceMorphism C B s t
                  ; _≈_ = λf g ↦ SliceMorphism.m f ≈
                                  SliceMorphism.m g
                  ; proofs of laws }
    ; proofs of laws }
record Span C L R : Set _ where
  constructor span
  field
    M : Object
    l  : M ⇒ L
    r  : M ⇒ R
record SpanMorphism C L R (s t : Span C L R) : Set _ where
  constructor spanMorphism
  field
    m : Span.M s ⇒ Span.M t
    triangle-l : Span.l t · m ≈ Span.l s
    triangle-r : Span.r t · m ≈ Span.r s
SpanCategory C L R : Category
SpanCategory C L R =
  record
    { Object     = Span C L R
    ; Morphism =
        λs t ↦ record
                  { Carrier = SpanMorphism C L R s t
                  ; _≈_ = λf g ↦ SpanMorphism.m f ≈
                                  SpanMorphism.m g
                  ; proofs of laws }
    ; proofs of laws }
```

Figure 4. Definitions of slice categories and span categories, where C : Category and B, L, R are objects of C.

- *Span categories* are similar: parametrised by two objects L and R, a span category has

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

  The Agda definitions are shown in the lower half of Figure 4, and are similar to those for slice categories.

- An object X in C is *terminal* if it satisfies the universal property that for every object Y there is a unique morphism from Y to X:

  ```
  Terminal C : Object → Set _
  Terminal C X =
    (Y : Object) → Σ[f : Y ⇒ X] Unique (Morphism Y X) f
  ```

  where uniqueness is defined relative to a setoid:

  ```
  Unique : (S : Setoid) → Carrier_S → Set _
  Unique S x = (y : Carrier_S) → x ≈_S y
  ```

- A *product* of two objects X and Y in C is then a Span $C\,X\,Y$ that is terminal in *SpanCategory* $C\,X\,Y$:

  ```
  Product C X Y : Span C X Y → Set _
  Product C X Y = Terminal (SpanCategory C X Y)
  ```

- A *pullback* of two slices f, g : Slice $C\,X$ is a product of f and g in *SliceCategory* $C\,X$: Define the type of *squares* based on f and g as

  ```
  Square C f g : Set _
  Square C f g = Span (SliceCategory C X) f g
  ```

  or diagrammatically,

  $$\begin{tikzcd} & W \arrow[dl, "l"'] \arrow[dr, "r"] \arrow[dd] & \\ Y \arrow[dr] & & Z \arrow[dl] \\ f & X & g \end{tikzcd}$$

  which is the same as

  $$\begin{tikzcd} f & & g \\ Y \arrow[d] & W \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & Z \arrow[d] \\ X \arrow[r, phantom, "="] & X \arrow[r, phantom, "="] & X \end{tikzcd}$$

  In a square q, we will refer to the object Slice.T (Span.$M\,q$), i.e., the node W in the diagrams above, as the *vertex* of q. A pullback of f and g is then a square based on f and g that satisfies

  ```
  Pullback C f g : Square C f g → Set _
  Pullback C f g = Product (SliceCategory C X) f g
  ```

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

  ```
  SquareCategory C f g : Category
  SquareCategory C f g =
    SpanCategory (SliceCategory C X) f g
  ```

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

The most important category-theoretic fact that we will use in this paper is that the vertices of any two pullbacks of the same slices are isomorphic. Define the type of isomorphisms between two objects X and Y in C as

```
record Iso C X Y : Set _ where
  field
    to   : X ⇒ Y
    from : Y ⇒ X
    from-to-inverse : from · to ≈ id
    to-from-inverse : to · from ≈ id
```

(The isomorphism relation _≅_ we used in Section 2 is formally defined as Iso 𝔽UN.) Then we can formulate the following lemma:

Lemma 1. *If* p, q : *Square* $C\,f\,g$ *are both pullbacks, then we have an isomorphism*

```
Iso C (Slice.T (Span.M p)) (Slice.T (Span.M q))
```

4. Categorical organisation of the ornament–refinement framework

We proceed to organise the ornament–refinement framework under several concrete categories and functors, aiming to clarify the overall structure of the framework, and then derive useful pullback properties from parallel composition of ornaments.

4.1 The category of type families and refinement families

We will see that the category 𝔽REF of type families and refinement families has a very close relationship to the category 𝔽AM. An object in 𝔽REF is an indexed family of sets as in 𝔽AM, and a morphism from (J,Y) to (I,X) consists of a function $e : J \to I$ on the indices and a refinement family of type *FRefinement* $e\,X\,Y$. As for the equivalence on morphisms, it suffices to use extensional equality on the index functions and componentwise equivalence on refinement families, where the equivalence on refinements is defined to be extensional equality on their forgetful functions (extracted by Refinement.*forget*). In Agda:

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

Two facts support our choice of refinement equivalence: (i) under this definition, if two refinements are equivalent, then their promotion predicates are pointwise isomorphic, i.e., we have

```
forget-iso :
  {X Y : Set} (r s : Refinement X Y) →
  (Refinement.forget r ≐ Refinement.forget s) →
  ∀ x → Refinement.P r x ≅ Refinement.P s x
```

and (ii) we get a forgetful functor FREFF : Functor 𝔽REF 𝔽AM which is identity on objects and componentwise Refinement.*forget* on morphisms, the latter respecting equivalence automatically.

```
FREFF : Functor 𝔽REF 𝔽AM
FREFF = record
  { object   = λ (I,X) ↦ I,X
  ; morphism =
      λ (e,rs) ↦ e, (λ {j} ↦ Refinement.forget (rs (ok j)))
  ; proofs of laws }
```

Note that a refinement family from $X : I \to$ Set to $Y : J \to$ Set is deliberately cast as a morphism in the opposite direction from (J,Y) to (I,X), so FREFF remains a familiar covariant functor rather than a contravariant one. Think of this as suggesting the direction of the forgetful functions of refinements.

The above discussion suggests that the essential ingredient of a refinement is just its forgetful function. Indeed, from any function we can construct a *canonical refinement*:

```
canonRef : {X Y : Set} → (Y → X) → Refinement X Y
canonRef {X} {Y} f = record
  { P = λx ↦ Σ[y : Y] f y ≡ x
  ; i = record { to   = ⟨f,⟨(λy ↦ y),(λy ↦ refl)⟩⟩
               ; from = proj₁ ∘ proj₂
               ; proofs of laws }}
```

(The operator $\langle\_,\_\rangle$ is defined by $\langle g,h\rangle = \lambda x \mapsto (g\,x, h\,x)$.) The canonical promotion predicate is very simplistic: to promote some $x : X$ to type Y, we are required to supply a complete $y : Y$ such that x can be recovered from y (rather than only the necessary information that augments x to an element of Y). Any refinement r : Refinement $X\,Y$ is equivalent to *canonRef* (Refinement.*forget* r), so by *forget-iso* we have

```
Refinement.P r x ≅ Σ[y : Y] Refinement.forget r y ≡ x
```

$$\begin{tikzcd}
\mathbb{O}\textsc{rn} \arrow[rr, bend left=30, "\textsc{Ind}"] \arrow[r, bend right, "\textsc{RSem}"'] & \mathbb{F}\textsc{Ref} \arrow[r, bend left, "\textsc{FRefF}"] & \mathbb{F}\textsc{am} \arrow[l, bend left, "\textsc{FRefC}"] \arrow[r, bend left, "\textsc{FamF}"] \arrow[r, bend right, "\textsc{FamI}"'] & \mathbb{F}\textsc{un}
\end{tikzcd}$$

Object	$\Sigma[I : \mathsf{Set}]\ \mathsf{Desc}\,I$	$\Sigma[I : \mathsf{Set}]\ I \to \mathsf{Set}$	$\Sigma[I : \mathsf{Set}]\ I \to \mathsf{Set}$	Set
$\_\Rightarrow\_$	$\lambda(J,E)\ (I,D) \mapsto$ $\Sigma[e : J \to I]\ \mathsf{Orn}\ e\ D\ E$	$\lambda(J,Y)\ (I,X) \mapsto$ $\Sigma[e : J \to I]\ \mathit{FRefinement}\ e\ X\ Y$	$\lambda(J,Y)\ (I,X) \mapsto$ $\Sigma[e : J \to I]\ Y \rightrightarrows (X \circ e)$	$\lambda A\ B \mapsto A \to B$

Figure 5. Categories (whose sets of objects and morphisms are listed below) and functors for the ornament–refinement framework.

for all $x : X$. That is, a promotion predicate is always pointwise isomorphic to the canonical promotion predicate. Thus all the refinement mechanism provides is a convenient way of expressing intensional (representational) optimisations of the canonical promotion predicate — extensionally, $\mathbb{F}$REF is no more powerful than $\mathbb{F}$AM. This is reflected in the existence of a functor FREFC : Functor $\mathbb{F}$AM $\mathbb{F}$REF, whose object part is identity and whose morphism part is componentwise *canonRef*:

```
FRefC : Functor 𝔽Am 𝔽Ref
FRefC = record
  { object    = λ(I,X) ↦ I,X
  ; morphism  = λ(e,u) ↦ e,(λ(ok j) ↦ canonRef (u {j}))
  ; proofs of laws }
```

FREFC is strictly inverse to FREFF, forming an isomorphism (not merely an equivalence) of categories between $\mathbb{F}$REF and $\mathbb{F}$AM.

4.2 The category of descriptions and ornaments

The category $\mathbb{O}$RN has objects of type $\Sigma[I : \mathsf{Set}]\ \mathsf{Desc}\ I$, i.e., descriptions paired with index sets, and morphisms from (J,E) to (I,D) of type $\Sigma[e : J \to I]\ \mathsf{Orn}\ e\ D\ E$, i.e., ornaments paired with index erasure functions. We also need to devise an equivalence on ornaments

```
OrnEq :
  {I J : Set} {e e′ : J → I} {D : Desc I} {E : Desc J} →
  Orn e D E → Orn e′ D E → Set
```

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

```
OrnEq-forget :
  {I J : Set} {e e′ : J → I} {D : Desc I} {E : Desc J}
  (O : Orn e D E) (P : Orn e′ D E) → OrnEq O P →
  (e ≐ e′) × (∀ {j} → forget O {j} ≃ forget P {j})
```

We omit the detail of OrnEq from the paper (which depends on the detail of the universe of ornaments). Morphism composition is sequential composition, and there is a family of *identity ornaments*

```
idOrn : {I : Set} {D : Desc I} → Orn (λi ↦ i) D D
```

such that *idOrn* $\{I\}$ $\{D\}$ simply expresses that D is identical to itself. Unsurprisingly, the identity ornaments serve as identity of sequential composition. To summarise:

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

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

```
Ind : Functor 𝕆rn 𝔽am
Ind = record { object    = λ(I,D) ↦ I,μ D
             ; morphism  = λ(e,O) ↦ e,forget O
             ; proofs of laws }
```

4.3 Pullback properties for parallel composition

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

$$\begin{tikzcd}
e \bowtie f, \lfloor O \otimes P \rfloor \arrow[r, "\pi_2{,}\, \mathit{diffOrn\text{-}r}\ O\ P"] \arrow[d, "\pi_1{,}\, \mathit{diffOrn\text{-}l}\ O\ P"'] \arrow[dr, "\mathit{pull}{,}\, \lceil O \otimes P \rceil" description] & K,F \arrow[d, "f{,}P"] \\
J,E \arrow[r, "e{,}O"'] & I,D
\end{tikzcd}$$

This square is encoded in Agda as

```
pc-square : Square 𝕆rn (slice (J,E) (e,O)) (slice (K,F) (f,P))
pc-square = span (slice (e ⋈ f,⌊O ⊗ P⌋) (pull,⌈O ⊗ P⌉))
                 (sliceMorphism (π₁,diffOrn-l O P) { }₁)
                 (sliceMorphism (π₂,diffOrn-r O P) { }₂)
```

where goal 1 has type OrnEq $(O \odot \mathit{diffOrn\text{-}l}\ O\ P)\ \lceil O \otimes P \rceil$ and goal 2 has type OrnEq $(P \odot \mathit{diffOrn\text{-}r}\ O\ P)\ \lceil O \otimes P \rceil$, both of which can be discharged.[2] The pullback property of *pc-square*, i.e., *Pullback* $\mathbb{O}$RN _ _ *pc-square*, is not too useful by itself, though: $\mathbb{O}$RN is quite a restricted category, so a universal property established in $\mathbb{O}$RN has limited applicability. Instead, we are more interested in the pullback property of the image of the above square under IND in $\mathbb{F}$AM, which is stated in the follow theorem.

[2] Since the structure of Agda terms like *pc-square* can be reconstructed from commutative diagrams and the categorical definitions, in the rest of the paper we will present only the commutative diagrams and omit the underlying Agda terms.

Theorem 1 (pullback property for parallel composition in $\mathbb{F}$AM)**.** *If O : $\mathsf{Orn}\;e\;D\;E$ and P : $\mathsf{Orn}\;f\;D\;F$ where D : $\mathsf{Desc}\;I$, E : $\mathsf{Desc}\;J$, and F : $\mathsf{Desc}\;K$, then the following square in $\mathbb{F}$AM is a pullback.*

$$\begin{tikzcd}
e \bowtie f, \mu\;\lfloor O \otimes P \rfloor \arrow[r, "\pi_2{,}\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;P)"] \arrow[d, "\pi_1{,}\mathit{forget}\;(\mathit{diffOrn\text{-}l}\;O\;P)"'] \arrow[dr, "\mathit{pull}{,}\mathit{forget}\;\lceil O \otimes P \rceil" description] \arrow[dr, phantom, "\lrcorner", very near start] & K, \mu\;F \arrow[d, "f{,}\mathit{forget}\;P"] \\
J, \mu\;E \arrow[r, "e{,}\mathit{forget}\;O"'] & I, \mu\;D
\end{tikzcd}$$

The proof of the universal property boils down to, very roughly speaking, construction of an inverse to

$$\langle \mathit{forget}\;(\mathit{diffOrn\text{-}l}\;O\;P), \mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;P)\rangle$$

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

Corollary 1 (pullback property for parallel composition in $\mathbb{F}$UN)**.** *If O : $\mathsf{Orn}\;e\;D\;E$ and P : $\mathsf{Orn}\;f\;D\;F$ where D : $\mathsf{Desc}\;I$, E : $\mathsf{Desc}\;J$, and F : $\mathsf{Desc}\;K$, then the following square in $\mathbb{F}$UN is a pullback.*

$$\begin{tikzcd}
\Sigma\;(e \bowtie f)\;(\mu\;\lfloor O \otimes P \rfloor) \arrow[r, "\pi_2 * \mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;P)"] \arrow[d, "\pi_1 * \mathit{forget}\;(\mathit{diffOrn\text{-}l}\;O\;P)"'] \arrow[dr, "\mathit{pull} * \mathit{forget}\;\lceil O \otimes P \rceil" description] \arrow[dr, phantom, "\lrcorner", very near start] & \Sigma\;K\;(\mu\;F) \arrow[d, "f * \mathit{forget}\;P"] \\
\Sigma\;J\;(\mu\;E) \arrow[r, "e * \mathit{forget}\;O"'] & \Sigma\;I\;(\mu\;D)
\end{tikzcd}$$

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

$$\mathbb{O}\text{RN} \xrightarrow{\text{IND}} \mathbb{F}\text{AM} \xrightarrow{\text{FREFC}} \mathbb{F}\text{REF}$$

The resulting refinements would then use canonical promotion predicates. However, the whole point of incorporating $\mathbb{O}$RN in the framework is that we can construct an alternative functor RSEM directly from $\mathbb{O}$RN to $\mathbb{F}$REF. The functor RSEM is extensionally equal to the above composite functor, but intensionally very different. Its object part still takes the least fixed point of a description, but its morphism part is the refinement semantics of ornaments given in Section 2.6, whose promotion predicates have a more efficient representation.

```
RSEM : Functor 𝕆RN 𝔽REF
RSEM = record
  { object    = λ(I,D) ↦ I, μ D
  ; morphism  =
      λ(e,O) ↦ e, (λ(ok j) ↦ record { P = OptP O (ok j)
                                      ; i = { }3 })
  ; proofs of laws }
```

We will give goal 3, i.e., the ornamental promotion isomorphisms, a new construction in the next section.

5. Reconstruction of the ornamental promotion and modularity isomorphisms

The morphism part of the functor RSEM : $\mathsf{Functor}$ $\mathbb{O}$RN $\mathbb{F}$REF translates ornaments into refinements that use the optimised predicates, which are defined via parallel composition, so the pullback properties for parallel composition hold for the optimised predicates. The natural step to take, then, is to construct the ornamental promotion isomorphisms using the pullback properties — this we do in the proof of Theorem 2 below. Even more closely related are the modularity isomorphisms, which are about parallel composition and optimised predicates. They, too, can be constructed from the pullback properties for parallel composition, which is done in the proof of Theorem 3.

We restate the ornamental promotion isomorphisms as the following theorem.

Theorem 2 (ornamental promotion isomorphisms)**.** *For any ornament O : $\mathsf{Orn}\;e\;D\;E$ where D : $\mathsf{Desc}\;I$ and E : $\mathsf{Desc}\;J$, we have*

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

for all j : J.

Proof. Since the optimised predicates $\mathit{OptP}\;O$ are defined by parallel composition of O and the singleton ornament $S = \mathit{singOrn}\;D$, the conclusion of the theorem expand to

$$\mu\;E\;j \;\cong\; \Sigma[x : \mu\;D\;(e\;j)]\;\; \mu\;\lfloor O \otimes \lceil S \rceil \rfloor\;(\mathsf{ok}\;j, \mathsf{ok}\;(e\;j, x)) \tag{2}$$

How do we derive these isomorphisms from the pullback properties for parallel composition? It turns out that the pullback property in $\mathbb{F}$UN (Corollary 1) can help.

First, observe that we have the following pullback square:

$$\begin{tikzcd}
\Sigma\;J\;(\mu\;E) \arrow[r, "\langle e * \mathit{forget}\;O{,}\;\mathit{singleton} \circ \mathit{forget}\;O \circ \mathsf{proj}_2 \rangle"] \arrow[d, "\mathit{id}"'] \arrow[dr, "e * \mathit{forget}\;O" description] \arrow[dr, phantom, "\lrcorner", very near start] & \Sigma\;(\Sigma\;I\;(\mu\;D))\;(\mu\;\lfloor S \rfloor) \arrow[d, "\mathsf{proj}_1 * \mathit{forget}\;\lceil S \rceil"] \\
\Sigma\;J\;(\mu\;E) \arrow[r, "e * \mathit{forget}\;O"'] & \Sigma\;I\;(\mu\;D)
\end{tikzcd} \tag{3}$$

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

$$s \;=\; \mathsf{slice}\;(\Sigma\;(\Sigma\;I\;(\mu\;D))\;(\mu\;\lfloor S \rfloor))\;(\mathsf{proj}_1 * \mathit{forget}\;\lceil S \rceil)$$

behaves like a "multiplicative unit": any (compatible) slice s' alone gives rise to a product of s and s'. As a consequence, we have the bottom-left type $\Sigma\;J\;(\mu\;E)$ as the vertex of the pullback. This pullback square is based on the same slices as the one in Corollary 1 with P substituted by $\lceil S \rceil$, so by Lemma 1 we obtain an isomorphism

$$\Sigma\;J\;(\mu\;E) \;\cong\; \Sigma\;(e \bowtie \mathsf{proj}_1)\;(\mu\;\lfloor O \otimes \lceil S \rceil \rfloor) \tag{4}$$

To get from (4) to (2), we need to look more closely into the construction of (4). The right-to-left direction of (4) is obtained by applying the universal property of (3) to the square in Corollary 1 (with P substituted by $\lceil S \rceil$), so it is the unique mediating morphism m that makes the following diagram commute:

$$\begin{tikzcd}
 & \Sigma\;(e \bowtie \mathsf{proj}_1)\;(\mu\;\lfloor O \otimes \lceil S \rceil \rfloor) \arrow[dl, "\pi_1 * \mathit{forget}\;(\mathit{diffOrn\text{-}l}\;O\;P)"'] \arrow[dr, "\pi_2 * \mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;P)"] \arrow[dd, dashed, "m"'] & \\
\Sigma\;J\;(\mu\;E) & & \Sigma\;(\Sigma\;I\;(\mu\;D))\;(\mu\;\lfloor S \rfloor) \\
 & \Sigma\;J\;(\mu\;E) \arrow[ul, "\mathit{id}"] \arrow[ur, "\langle e * \mathit{forget}\;O{,}\;\mathit{singleton} \circ \mathit{forget}\;O \circ \mathsf{proj}_2 \rangle"'] &
\end{tikzcd}$$

From the left commuting triangle, we see that, extensionally, the morphism m is just $\pi_1 * \mathit{forget}\;(\mathit{diffOrn\text{-}l}\;O\;P)$. This leads us to the following general lemma:

Lemma 2. *If there is an isomorphism*

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

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

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

for all k : K.

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

Specialising Lemma 2 for (4), we get

$$\mu\;E\;j \;\cong\; \Sigma[jix : \pi_1{}^{-1}\,j]\;\;\mu\;\lfloor O \otimes \lceil S\rceil\rfloor\;(\mathit{und}\;jix) \tag{5}$$

for all $j : J$. Finally, observe that a canonical element of type $\pi_1{}^{-1}\,j$ must be of the form $\mathsf{ok}\;(\mathsf{ok}\;j, \mathsf{ok}\;(e\;j,x))$ for some $x : \mu\;D\;(e\;j)$, so we perform a change of variables for the summation, turning the right-hand side of (5) into

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

and arriving at (2). □

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

The other important family of isomorphisms we should consider is the modularity isomorphisms.

Theorem 3 (modularity isomorphisms)**.** *Suppose that there are descriptions $D : \mathsf{Desc}\;I$, $E : \mathsf{Desc}\;J$ and $F : \mathsf{Desc}\;K$, and ornaments $O : \mathsf{Orn}\;e\;D\;E$, and $P : \mathsf{Orn}\;f\;D\;F$. Then we have*

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

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

Proof. The conclusion of the theorem expands to

$$\begin{aligned}&\mu\;\lfloor\lceil O \otimes P\rceil \otimes \lceil S\rceil\rfloor\;(\mathsf{ok}\;(j,k), \mathsf{ok}\;(i,x))\\ &\quad\cong\; \mu\;\lfloor O \otimes \lceil S\rceil\rfloor\;(j, \mathsf{ok}\;(i,x)) \;\times\; \mu\;\lfloor P \otimes \lceil S\rceil\rfloor\;(k, \mathsf{ok}\;(i,x))\end{aligned} \tag{6}$$

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

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

and

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

both indexed by $\mathit{pull} \bowtie \mathsf{proj}_1$ where pull has type $e \bowtie f \;\to\; I$ and proj_1 has type $\Sigma\;I\;X \;\to\; I$. This is just an isomorphism in $\mathbb{F}$AM between $(\mathit{pull} \bowtie \mathsf{proj}_1, M)$ and $(\mathit{pull} \bowtie \mathsf{proj}_1, N)$ whose index part (i.e., the isomorphism obtained under the functor FAMI) is identity. Thus we seek to prove that both $(\mathit{pull} \bowtie \mathsf{proj}_1, M)$ and $(\mathit{pull} \bowtie \mathsf{proj}_1, N)$ are vertices of pullbacks based on the same slices.

Let us look at $(\mathit{pull} \bowtie \mathsf{proj}_1, N)$ first. For fixed i, j, k, and x, the set $N\;(\mathsf{ok}\;(j,k), \mathsf{ok}\;(i,x))$ along with the cartesian projections is a product, which trivially extends to a pullback since there is a forgetful function from each of the two component sets to the *singleton* set $\mu\;\lfloor S\rfloor\;(i,x)$, as shown in the following diagram:

$$\begin{tikzcd}
N\;(\mathsf{ok}\;(j,k), \mathsf{ok}\;(i,x)) \arrow[r, "\mathsf{proj}_2"] \arrow[d, "\mathsf{proj}_1"'] \arrow[dr, phantom, very near start, "\lrcorner"] & \mu\;\lfloor P \otimes \lceil S\rceil\rfloor\;(k, \mathsf{ok}\;(i,x)) \arrow[d, "\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;P\;\lceil S\rceil)"] \\
\mu\;\lfloor O \otimes \lceil S\rceil\rfloor\;(j, \mathsf{ok}\;(i,x)) \arrow[r, "\mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;\lceil S\rceil)"'] & \mu\;\lfloor S\rfloor\;(i,x)
\end{tikzcd}$$

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

$$\begin{tikzcd}
\mathit{pull} \bowtie \mathsf{proj}_1, N \arrow[r, "\_, \mathsf{proj}_2"] \arrow[d, "\_, \mathsf{proj}_1"'] \arrow[dr, phantom, very near start, "\lrcorner"] & f \bowtie \mathsf{proj}_1, \mu\;\lfloor P \otimes \lceil S\rceil\rfloor \arrow[d, "\pi_2, \mathit{forget}\;(\mathit{diffOrn\text{-}r}\;P\;\lceil S\rceil)"] \\
e \bowtie \mathsf{proj}_1, \mu\;\lfloor O \otimes \lceil S\rceil\rfloor \arrow[r, "\pi_2, \mathit{forget}\;(\mathit{diffOrn\text{-}r}\;O\;\lceil S\rceil)"'] & \Sigma\;I\;(\mu\;D), \mu\;\lfloor S\rfloor
\end{tikzcd} \tag{7}$$

Next we prove that $(\mathit{pull} \bowtie \mathsf{proj}_1, M)$ is also the vertex of a pullback based on the same slices as (7). This second pullback arises as a consequence of the following lemma.

Lemma 3. *In any category, consider the objects X, Y, their product $X \Leftarrow X \boxtimes Y \Rightarrow Y$, and products of each of the three objects X, Y, and $X \boxtimes Y$ with an object Z. (All the projections are shown as solid arrows in the diagram below). Then $(X \boxtimes Y) \boxtimes Z$ is the vertex of a pullback of the two projections $X \boxtimes Z \Rightarrow Z$ and $Y \boxtimes Z \Rightarrow Z$.*

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

We again intend to view a pullback as a product of slices, and instantiate Lemma 3 in *SliceCategory* $\mathbb{F}$AM $(I, \mu\;D)$, substituting all the objects by slices consisting of relevant ornamental forgetful functions in (6). The substitutions are as follows:

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

where $X \boxtimes Y$, $X \boxtimes Z$, $Y \boxtimes Z$, and $(X \boxtimes Y) \boxtimes Z$ indeed give rise to products in *SliceCategory* $\mathbb{F}$AM $(I, \mu\;D)$, i.e., pullbacks in $\mathbb{F}$AM, by instantiating Theorem 1. What we get out of this instantiation of the lemma is a pullback in *SliceCategory* $\mathbb{F}$AM $(I, \mu\;D)$ rather than $\mathbb{F}$AM. This is easy to fix, since there is a forgetful functor from any *SliceCategory C B* to *C* whose object part is Slice.*T*, and it is pullback-preserving. We thus get a pullback in $\mathbb{F}$AM which is based on the same slices as (7) and has vertex $(\mathit{pull} \bowtie \mathsf{proj}_1, M)$.

Having the two pullbacks, by Lemma 1 we get an isomorphism in $\mathbb{F}$AM between $(\mathit{pull} \bowtie \mathsf{proj}_1, M)$ and $(\mathit{pull} \bowtie \mathsf{proj}_1, N)$, whose index part can be shown to be identity, so there are componentwise isomorphisms between M and N in $\mathbb{F}$UN, arriving at (6). □

6. Discussion

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

- the least fixed-point operation on descriptions (the object part of the functor),
- the forgetful functions derived from ornaments (the morphism part of the functor),

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

Moreover, a categorical pullback structure emerges from the framework, enabling new constructions of the ornamental promotion isomorphisms and the modularity isomorphisms on a more abstract level. The new constructions of the isomorphisms offer more insights and are easier to understand (compared to the implementations in our previous work [10] using datatype-generic induction): after establishing the pullback properties of parallel composition, at the root of the ornamental promotion isomorphisms is the intuition that singleton ornaments do not add information, and the modularity isomorphisms stem from the fact that the *pointwise* conjunction of optimised predicates trivially extends to a pullback. Also, the categorical constructions are impervious to change of representation of the universes; modification of the universes only affects constructions logically prior to Theorem 1. (This statement is empirically verified: we really had to change the implementation of the universes once after carrying out the categorical constructions.)

Dagand and McBride [7] provided a categorical treatment of ornaments using fibred category theory, which is quite independent of our work, though. They established correspondences between descriptions and polynomial functors [9] and between ornaments and cartesian morphisms, and sketched how several operations on ornaments correspond to certain categorical notions (including pullbacks), whereas we skip giving a functorial semantics to descriptions and ornaments and directly translate them into inductive families and componentwise functions, which serves our purpose well. A more significant methodological difference, however, is that Dagand and McBride distinguish "software" (e.g., descriptions) and "mathematics" (e.g., polynomial functors) and then make a connection between them, so they can use mathematical notions as inspiration for software constructs. We believe that a further step should be taken: rather than merely making a connection between software artifacts that have the necessary level of detail and mathematical objects that possess desired abstract properties, we should design the software artifacts such that they satisfy the abstract properties themselves, so the detail of the artifacts can be effectively managed by reasoning in terms of the abstract properties. Consequently, we use category theory as an organising principle on top of type theory — we think of category theory as providing a set of abstractions that are internalised in type theory, rather than being an independent formalism. Our version of ornaments inherently exhibit a categorical structure, and by reasoning in terms of this categorical structure, we are able to tame the complexity of ornaments, which exists in order to provide representational optimisation.

Our presentation of the categorical organisation has its origin in the old revelation that "software" and "mathematics" should be treated uniformly as one and the same entity [12]. Hence Agda is used throughout the paper as a uniform language for expressing both programming constructs and categorical notions. In particular, all commutative diagrams are no more than compact forms of equivalences that can be formally expressed in Agda. We believe that basing all formal entities on type theory yields a clean and precise presentation, although Agda does not provide enough support for this yet. Syntactically, Agda's implicit parameters and mixfix operators are already helpful in suppressing noise and introducing familiar notations, but more deeply, intensionality remains a great obstacle to formulating straightforward and readable proofs. For non-trivial proofs like the ones in Section 5, our presentation has to deviate from the actual Agda proofs to hide the detail for overcoming intensionality. We look forward to advances in theoretical foundations (such as the univalent foundations project [19]) and syntactic support for presenting type-theoretic proofs that are readable by both human and machine, delivering both intuition and precision at the same time.

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] G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

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

[4] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, 2004.

[5] 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.

[6] P.-É. Dagand and C. McBride. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM, 2012.

[7] P.-É. Dagand and C. McBride. A categorical treatment of ornaments. To appear in *Logic in Computer Science*, 2013.

[8] P.-É. Dagand and C. McBride. Elaborating inductive definitions. In *Journées Francophones des Langages Applicatifs*, JFLA'13. INRIA, 2013.

[9] N. Gambino and J. Kock. Polynomial functors and polynomial monads. arXiv:0906.4931, 2010.

[10] H.-S. Ko and J. Gibbons. Modularising inductive families. *Progress in Informatics*, 10:65–88, 2013. doi:10.2201/NiiPi.2013.10.5.

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

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

[13] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

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

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

[16] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of Programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579, 2009.

[17] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

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

[19] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton, 2013. URL `http://homotopytypetheory.org/book/`.