Chapter 4

Categorical organisation of the ornament–refinement framework

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

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

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

  -- definition in Figure 4.4

  Correspondingly, we expect that

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

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

- While parallel composition of ornaments (**??**) has a sensible definition, it is defined by case analysis at the microscopic level of individual fields. Such a microscopic definition is difficult to comprehend, and so are any subsequent definitions and proofs. It is desirable to have a macroscopic characterisation of parallel composition, so the nature of parallel composition is immediately

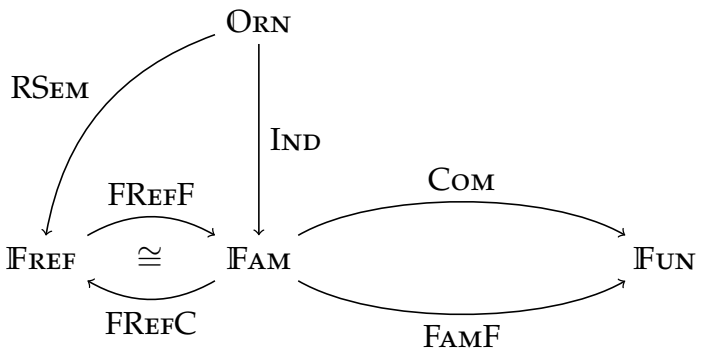


Figure 4.1 Categories and functors for the ornament–refinement framework.

clear, and subsequent definitions and proofs can be done in a more abstract manner.

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

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

- Categories and functors are abstractions for compositional structures and structure-preserving maps between them. Facts about translations between ornaments, refinements, and functions can thus be neatly organised under the categorical language (Section 4.1). The categories and functors used in this chapter are summarised in Figure 4.1.
- Parallel composition merges two compatible ornaments and does nothing more; in other words, it computes the least informative ornament that contains the information of both ornaments. Characterisation of such **universal constructions** is a speciality of category theory; in our case, parallel composition can be shown to be a categorical **pullback** (Section 4.2).

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

Section 4.4 concludes with some discussion.

4.1 Categories and functors

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

4.1.1 Basic definitions

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

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

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

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

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

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

which has left and right identities and is associative.

Syntactic remark (*universe polymorphism*)**.** Many definitions in this chapter (like Graph above) employ Agda's **universe polymorphism** [Harper and Pollack, 1991], so the definitions can be instantiated at suitable levels of the Set hierarchy as needed. (For example, the type of Fun-*graph* is implicitly instantiated as Graph $\{1\}$ $\{0\}$, since Set is of type Set_1 and any $A \to B$ (where A, B : Set) are of type Set ($= \mathsf{Set}_0$), and Graph $\{1\}$ $\{0\}$ itself is of type Set_2, whose level is computed by taking the successor of the maximum of the two level arguments.) We will give the first few universe-polymorphic definitions with full detail about their levels, but will later suppress the syntactic noise wherever possible. □

Before we move on to the definition of categories, though, special attention must be paid to equality on morphisms, which is usually coarser than definitional equality — in Fun, for example, it is necessary to identify functions up to extensional equality (so uniqueness of morphisms in universal properties would make sense). As stated in **??**, such equalities need to be explicitly managed in Agda's intensional setting, and one way is to use **setoids** [Barthe et al., 2003] — sets with an explicitly specified equivalence relation — to represent sets of morphisms. Subsequently, functions defined between setoids need to be proved to respect the equivalences. The type of setoids can be defined as a record which contains a carrier set, an equivalence relation on the set, and the three laws for the equivalence relation:

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

Proofs of the three laws are omitted from the presentation.

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

Now we can define the category 𝔽un of sets and functions as

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

Another important category that we will make use of is 𝔽am (Figure 4.3), the category of indexed families of sets and indexed families of functions, which is useful for talking about componentwise structures. An object in 𝔽am has type $\Sigma[\, I : \mathsf{Set} \,]\ I \to \mathsf{Set}$, i.e., it is a set I and a family of sets indexed by I (forming this Σ-type requires a universe-polymorphic revision of the definition of Σ); a morphism from $(J\ ,\ Y)$ to $(I\ ,\ X)$ is a function $e : J \to I$ and a family of functions from $Y\ j$ to $X\ (e\ j)$ for each $j : J$. Morphism composition is componentwise composition, and morphism equivalence is defined to be componentwise extensional equality. (The morphism equivalence is formulated

```
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 : Object} → (X ⇒ Y) → (X ⇒ Y) → Set n
  _≈_ {X} {Y} = Setoid._≈_ (Morphism X Y)
  field
    _·_   : {X Y Z : Object} → (Y ⇒ Z) → (X ⇒ Y) → (X ⇒ Z)
    id    : {X : Object} → (X ⇒ X)
    id-l  : {X Y : Object} (f : X ⇒ Y) → id · f ≈ f
    id-r  : {X Y : Object} (f : X ⇒ Y) → f · id ≈ f
    assoc : {X Y Z W : Object} (f : Z ⇒ W) (g : Y ⇒ Z) (h : X ⇒ Y) →
            (f · g) · h ≈ f · (g · h)
    cong-l : {X Y Z : Object} {f g : Y ⇒ Z} (h : X ⇒ Y) → f ≈ g → f · h ≈ g · h
    cong-r : {X Y Z : Object} (h : Y ⇒ Z) {f g : X ⇒ Y} → f ≈ g → h · f ≈ h · g
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 : Object C} → X ⇒C Y → object X ⇒D object Y
    equiv-preserving : {X Y : Object C} {f g : X ⇒C Y} →
                       f ≈C g → morphism f ≈D morphism g
    id-preserving    : {X : Object C} → morphism (id C {X}) ≈D id D {object X}
    comp-preserving : {X Y Z : Object C} (f : Y ⇒C Z) (g : X ⇒C Y) →
                      morphism (f ·C g) ≈D (morphism f ·D morphism g)
```

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

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

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

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

Figure 4.3 (Partial) definitions of the categories 𝔽am, 𝔽ref, and 𝕆rn.

with the help of McBride's "John Major" heterogeneous equality $\_\cong\_$ [McBride, 1999] — the equivalence $\_\dot{\cong}\_$ is pointwise heterogeneous equality — since given $y : Y\ j$ for some $j : J$, the types of $u\ \{j\}\ y$ and $u'\ \{j\}\ y$ are not definitionally equal but only provably equal.)

Categories are graphs with a compositional structure, and **functors** are transformations between categories that preserve the compositional structure. The definition of functors is shown in Figure 4.2: a functor consists of two mappings, one on objects and the other on morphisms, where the morphism part preserves all structures on morphisms, including equivalence, identity, and composition. For example, we have two functors from $\mathbb{F}\textsc{am}$ to $\mathbb{F}\textsc{un}$, one summing components together

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

and the other extracting the index part.

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

Proofs of the functor laws are omitted from the presentation.

4.1.2 Categories and functors for refinements and ornaments

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

$e : J \to I$ on the indices and a refinement family of type FRefinement $e\ X\ Y$. As for the equivalence on morphisms, it suffices to use extensional equality on the index functions and componentwise extensional equality on refinement families, where extensional equality on refinements means extensional equality on their forgetful functions (extracted by Refinement.*forget*), which we have shown in **??** to be the core of refinements. Note that a refinement family from $X : I \to \mathsf{Set}$ to $Y : J \to \mathsf{Set}$ is deliberately cast as a morphism in the opposite direction from $(J\ ,\ Y)$ to $(I\ ,\ X)$; think of this as suggesting the direction of the forgetful functions of refinements. We can then define the following two functors, forming an isomorphism of categories between $\mathbb{F}\textsc{ref}$ and $\mathbb{F}\textsc{am}$:

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

  FRefF : Functor $\mathbb{F}\textsc{ref}$ $\mathbb{F}\textsc{am}$
  FRefF = **record**
    { *object* = *id*
    ; *morphism* = λ { $(e\ ,\ rs) \mapsto e\ ,\ (\lambda j \mapsto \mathsf{Refinement}.\mathit{forget}\ (rs\ (\mathsf{ok}\ j)))$ }
    ; proofs of laws }

  Note that FRefF remains a familiar covariant functor rather than a contravariant one because of our choice of morphism direction.

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

  FRefC : Functor $\mathbb{F}\textsc{am}$ $\mathbb{F}\textsc{ref}$
  FRefC = **record**
    { *object* = *id*
    ; *morphism* = λ { $(e\ ,\ u) \mapsto e\ ,\ \lambda$ { $(\mathsf{ok}\ j) \mapsto \mathit{canonRef}\ (u\ \{j\})$ } }
    ; proofs of laws }

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

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

```
𝔼-refl : (is : List I) → 𝔼 id is is
𝔼-refl []       = []
𝔼-refl (i :: is) = refl :: 𝔼-refl is

idROrn : (E : RDesc I) → ROrn id E E
idROrn (v is)   = v (𝔼-refl is)
idROrn (σ S E)  = σ[ s : S ] idROrn (E s)

idOrn : {I : Set} (D : Desc I) → Orn id D D
idOrn {I} D (ok i) = idROrn (D i)

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

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

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

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

- We need to devise an equivalence on ornaments

  $$\mathit{OrnEq} : \{I\ J : \mathsf{Set}\}\ \{e\ f : J \to I\}\ \{D : \mathsf{Desc}\ I\}\ \{E : \mathsf{Desc}\ J\} \to \mathsf{Orn}\ e\ D\ E \to \mathsf{Orn}\ f\ D\ E \to \mathsf{Set}$$

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

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

  The actual definition of *OrnEq* is deferred to **??**.

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

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

$$\begin{aligned}&\textsc{Ind} : \mathsf{Functor}\ \textsc{Orn}\ \textsc{Fam}\\ &\textsc{Ind} = \mathbf{record}\ \{\ \mathit{object} = \lambda\ \{\ (I\ ,\ D) \mapsto I\ ,\ \mu\ D\ \}\\ &\qquad\qquad ;\ \mathit{morphism} = \lambda\ \{\ (e\ ,\ O) \mapsto e\ ,\mathit{forget}\ O\ \}\\ &\qquad\qquad ;\ \text{proofs of laws}\ \}\end{aligned}$$

To translate Orn to Fref, a naive way is to use the composite functor FRefC ⋄ Ind : Functor Orn Fref, where composition $F \diamond G$ of functors F : Functor $D\ E$ and G : Functor $C\ D$ is defined by

$$\begin{aligned}&F \diamond G : \mathsf{Functor}\ C\ E\\ &F \diamond G = \mathbf{record}\ \{\ \mathit{object} = \mathit{object}\ F \circ \mathit{object}\ G\\ &\qquad\qquad ;\ \mathit{morphism} = \mathit{morphism}\ F \circ \mathit{morphism}\ G\\ &\qquad\qquad ;\ \text{proofs of laws}\ \}\end{aligned}$$

(We assume that there is an AGDA statement "**open** Functor" in scope throughout the dissertation, so Functor.*object* and Functor.*morphism* can simply be referred to as *object* and *morphism*.) The resulting refinements would then use the canonical promotion predicates. However, the whole point of incorporating ℂORN in the framework is that we can construct an alternative functor RSEM directly from ℂORN to 𝔽REF. The functor RSEM is extensionally equal to the above composite functor but intensionally different. While its object part still takes the least fixed point of a description, its morphism part is the refinement semantics of ornaments given in **??**, whose promotion predicates are the optimised predicates and have a more efficient representation.

```
RSem : Functor ℂOrn 𝔽Fam
RSem = record { object    = λ { (I , D) ↦ I , μ D    }
              ; morphism  = λ { (e , O) ↦ e , RSem O }
              ; proofs of laws }
```

4.1.3 Isomorphisms

So far the categorical organisation offers no obvious benefits, because we have not started talking about mapping purely categorical structures between categories, which is our main reason for employing the categorical language. One simplest example of such purely categorical structures is **isomorphisms**: the type of isomorphisms between two objects X and Y in a category C is defined by

```
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
```

(We assume that, by introducing the category C in the text, there is implicitly a statement **open** Category C, so $\_\Rightarrow\_$ refers to Category.$\_\Rightarrow\_$ C and so on.) The relation $\_\cong\_$ we have been using is formally defined as Iso 𝔽UN. Isomorphisms

are preserved by functors, i.e., for any F : Functor C D we have

Iso C X Y → Iso D (*object* F X) (*object* F Y)

which is proved by mapping all objects, morphisms, compositions, and equivalences in C appearing in the input isomorphism into D by F. This fact immediately tells us, for example, that when two ornaments are inverse to each other, so are their forgetful functions, by taking F = Ind.

Another useful class of purely categorical structures are introduced next.

4.2 Pullback properties of parallel composition

One of the great advantages of category theory is the ability to formulate the idea of **universal constructions** generically and concisely, which we will use to give parallel composition a useful macroscopic characterisation. An intuitive way to understand the idea of a universal construction is to think of it as a "strongly best" solution to some specification. More precisely: The specification is represented as a category whose objects are all possible solutions. A morphism from X to Y is evidence that Y is (non-strictly) "better" than X, and there can be more than one piece of such evidence. A "strongly best" solution is a **terminal object** in this category, meaning that it is "uniquely evidently better" than all objects in the category. Formally: an object Y in a category C is **terminal** when it satisfies the **universal property** that for every object X there is a unique morphism from X to Y, i.e., the setoid *Morphism* X Y has a unique inhabitant:

Terminal C Y : Set _
Terminal C Y = (X : *Object*) → *Singleton* (*Morphism* X Y)

where *Singleton* is defined by

Singleton : (S : Setoid) → Set _
Singleton S = Setoid.*Carrier* S × ((s t : Setoid.*Carrier* S) → s ≈$_S$ t)

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

```
terminal-iso C : (X Y : Object) → Terminal C X → Terminal C Y → Iso C X Y
terminal-iso C X Y tX tY =
  let f : X ⇒ Y
      f = outl (tY X)
      g : Y ⇒ X
      g = outl (tX Y)
  in record { to   = f
            ; from = g
            ; from-to-inverse = outr (tX X) (g · f) id
            ; to-from-inverse = outr (tY Y) (f · g) id }
```

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

Span categories and products

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

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

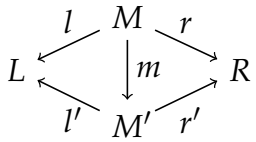
or diagrammatically:

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

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

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

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



where the two triangles are required to commute (i.e., *triangle-l* and *triangle-r* should hold). Thus a span s is more informative than another span s' when $\mathsf{Span}.M\ s$ is more informative than $\mathsf{Span}.M\ s'$ and the morphisms factorise appropriately. We can now form a category of spans over L and R:

```
SpanCategory C L R : Category
SpanCategory C L R = record
  { Object      = Span C L R
  ; Morphism =
      λ s s′ ↦ record
        { Carrier = SpanMorphism C L R s s′
        ; _≈_     = λ f g ↦ SpanMorphism.m f ≈ SpanMorphism.m g
        ; proofs of laws }
  ; proofs of laws }
```

Note that the equivalence on span morphisms is defined to be the equivalence on the mediating morphism in C, ignoring the two triangular commutativity

proofs. A product of L and R is then a terminal object in this category:

```
Product C L R : Span C L R → Set _
Product C L R = Terminal (SpanCategory C L R)
```

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

Slice categories

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

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

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

and

- morphisms: mediating morphisms giving rise to commutative triangles,

```
record SliceMorphism C B (s s′ : Slice C B) : Set _ where
  constructor _,_
  field
```

$$m \;:\; \mathsf{Slice}.T\ s \Rightarrow \mathsf{Slice}.T\ s'$$
$$\mathit{triangle} \;:\; \mathsf{Slice}.s\ s' \cdot m \;\approx\; \mathsf{Slice}.s\ s$$

or diagrammatically:

objects $\begin{array}{c} T \\ s \big\downarrow \\ B \end{array}$ and morphisms $\begin{array}{ccc} T & \xrightarrow{m} & T' \\ & {\scriptstyle s}\searrow \quad \swarrow{\scriptstyle s'} & \\ & B & \end{array}$

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

```
SliceCategory C B : Category
SliceCategory C B = record
  { Object     = Slice C B
  ; Morphism   =
      λ s s' ↦ record
        { Carrier  = SliceMorphism C B s s'
        ; _≈_      = λ f g ↦ SliceMorphism.m f ≈ SliceMorphism.m g
        ; proofs of laws }
  ; proofs of laws }
```

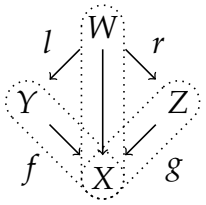
Objects in a slice category are thus morphisms with a common target, and when the ambient category is Orn, they are exactly the compatible ornaments that can be composed in parallel.

Pullbacks

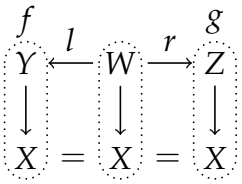
We have arrived at the characterisation of parallel composition as a product in a slice category on top of Orn. The composite term "product in a slice category" has become a multi-layered concept and can be confusing; to facilitate comprehension, we give several new definitions that can sometimes deliver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

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

since diagrammatically a square looks like



which is the same as



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

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

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

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

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

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

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

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

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

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

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

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

```
Pullback-preserving F :
  {B : Category.Object C} {f g : Slice C B} (s : Square C f g) →
  Pullback C f g s → Pullback D (object (SliceMap F) f) (object (SliceMap F) g)
                                (object (SquareMap F) s)
```

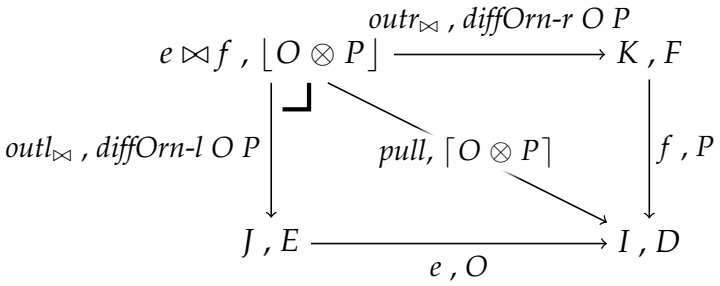
where

```
SliceMap    : (F : Functor C D) →
              Functor (SliceCategory C B) (SliceCategory D (object F B))
SquareMap   : (F : Functor C D) →
              Functor (SquareCategory C f g)
                      (SquareCategory D (object (SliceMap F) f)
                                        (object (SliceMap F) g))
```

are straightforward liftings of functors on ambient categories to functors on slice and square categories. Unlike isomorphisms, pullbacks are not preserved by all functors, but the functors Ind : Functor Orn Fam and Com : Functor Fam Fun are pullback-preserving, which we use below.

Parallel composition as a pullback

For any O : Orn e D E and P : Orn f D F where D : Desc I, E : Desc J, and F : Desc K, the following square in Orn is a pullback:



(4.2)

We assert that the square is a pullback by marking its vertex with "⌟". The Agda term for the square is

```
pc-square O P : Square Orn ((J , E) , (e , O)) ((K , F) , (f , P))
pc-square O P = ((e ⋈ f , ⌊O ⊗ P⌋) , (pull , ⌈O ⊗ P⌉)) ,
                ((outl⋈ , diffOrn-l O P) , { }0) ,
                ((outr⋈ , diffOrn-r O P) , { }1)
```

where Goal 0 has type $\mathit{OrnEq}\ (O \odot \mathit{diffOrn\text{-}l}\ O\ P)\ \lceil O \otimes P \rceil$ and Goal 1 has type $\mathit{OrnEq}\ (P \odot \mathit{diffOrn\text{-}r}\ O\ P)\ \lceil O \otimes P \rceil$, both of which can be discharged. Comparing the commutative diagram (4.2) and the Agda term *pc-square O P*, it should be obvious how concise the categorical language can be — the commutative diagram expresses the structure of the Agda term in a clean and visually intuitive way. Since terms like *pc-square O P* can be reconstructed from commutative diagrams and the categorical definitions, from now on we will present commutative diagrams as representations of the corresponding Agda terms and omit the latter. A proof sketch of (4.2) is deferred to **??**. The pullback property (4.2) by itself is not too useful in this chapter, though: Orn is a quite restricted category, so a universal property established in Orn has limited applicability. Instead, we are more interested in the following two pullback properties: the image of (4.2) under Ind in Fam:

$$\begin{array}{ccc} e \bowtie f\ ,\ \mu\ \lfloor O \otimes P \rfloor & \xrightarrow{\mathit{outr}_{\bowtie}\ ,\ \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ P)} & K\ ,\ \mu\ F \\ \Big\downarrow{\scriptstyle \mathit{outl}_{\bowtie}\ ,\ \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)} & \lrcorner\ \searrow^{\mathit{pull},\ \mathit{forget}\ \lceil O \otimes P \rceil} & \Big\downarrow{\scriptstyle f\ ,\ \mathit{forget}\ P} \\ J\ ,\ \mu\ E & \xrightarrow[e\ ,\ \mathit{forget}\ O]{} & I\ ,\ \mu\ D \end{array} \tag{4.3}$$

and the image of (4.3) under Com in Fun:

$$\begin{array}{ccc} \Sigma\ (e \bowtie f)\ (\mu\ \lfloor O \otimes P \rfloor) & \xrightarrow{\mathit{outr}_{\bowtie} * \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ P)} & \Sigma\ K\ (\mu\ F) \\ \Big\downarrow{\scriptstyle \mathit{outl}_{\bowtie} * \mathit{forget}\ (\mathit{diffOrn\text{-}l}\ O\ P)} & \lrcorner\ \searrow^{\mathit{pull} * \mathit{forget}\ \lceil O \otimes P \rceil} & \Big\downarrow{\scriptstyle f * \mathit{forget}\ P} \\ \Sigma\ J\ (\mu\ E) & \xrightarrow[e * \mathit{forget}\ O]{} & \Sigma\ I\ (\mu\ D) \end{array} \tag{4.4}$$

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

4.3 Consequences

Characterising parallel composition as a pullback immediately allows us to instantiate standard categorical results, like commutativity $(\_ , \lfloor O \otimes P \rfloor) \cong (\_ , \lfloor P \otimes O \rfloor)$ and associativity $(\_ , \lfloor \lceil O \otimes P \rceil \otimes Q \rfloor) \cong (\_ , \lfloor O \otimes \lceil P \otimes Q \rceil \rfloor)$ up to isomorphism in $\mathbb{O}\textsc{rn}$ (which can then be transferred by isomorphism preservation to $\mathbb{F}\textsc{am}$, for example). Our original motivation, on the other hand, is to implement the ornamental conversion isomorphisms and the modularity isomorphisms, which we carry out below.

4.3.1 The ornamental conversion isomorphisms

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

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

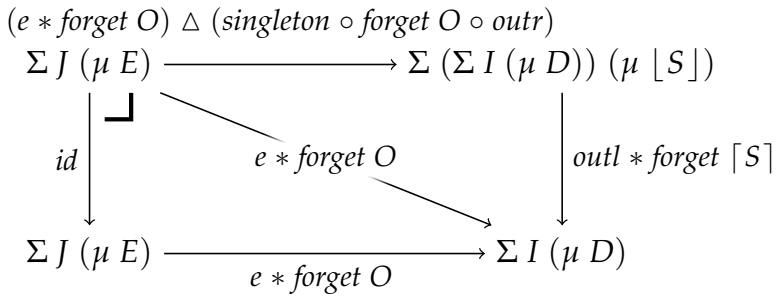
for all $j : J$. Since the optimised predicates $\mathsf{OptP}\ O$ are defined by parallel composition of O and the singleton ornamentation $S = \mathit{singletonOD}\ D$, the isomorphism expands to

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

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

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

following pullback square:



(4.6)

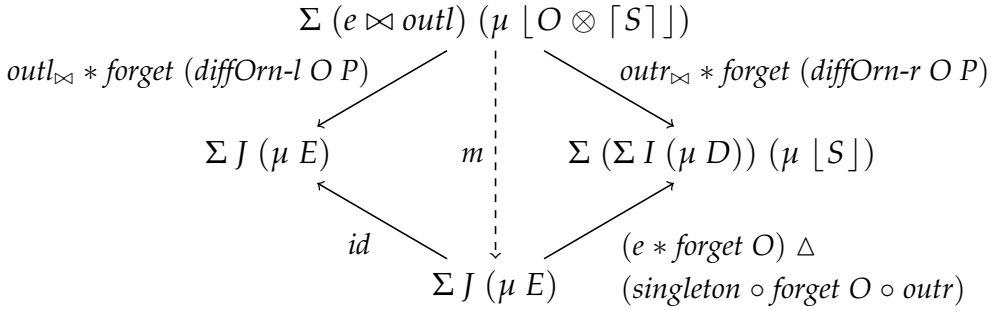
Since singleton ornamentation does not add information to a datatype, the vertical slice on the right-hand side

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

retains all information like an identity function, and thus behaves like a "multiplicative unit" (viewing pullbacks as products of slices): any (compatible) slice s' alone gives rise to a product of s and s'. In particular, we can use the bottom-left type $\Sigma\ J\ (\mu\ E)$ as the vertex of the pullback. This pullback square is over the same slices as the pullback square (4.4) with P substituted by $\lceil S \rceil$, so by (4.1) we obtain an isomorphism

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

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



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

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

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

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

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

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

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

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

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

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

  and arriving at (4.5).

4.3.2 The modularity isomorphisms

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

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

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

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

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

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

and

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

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

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

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

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

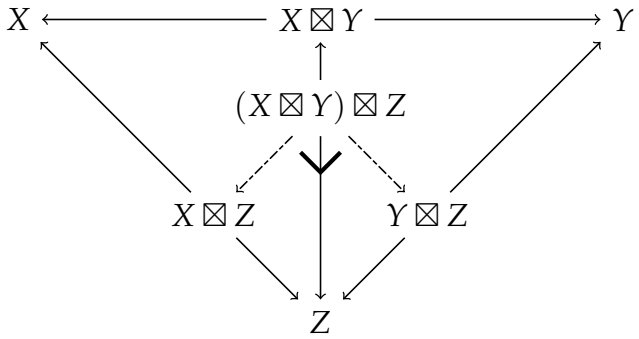
$$\begin{array}{ccc} N\ (\mathsf{ok}\ (j , k) , \mathsf{ok}\ (i , x)) & \xrightarrow{\mathit{outr}} & \mu \lfloor P \otimes \lceil S \rceil \rfloor\ (k , \mathsf{ok}\ (i , x)) \\ \Big\downarrow{\scriptstyle \mathit{outl}} & \lrcorner & \Big\downarrow{\scriptstyle \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ P\ \lceil S \rceil)} \\ \mu \lfloor O \otimes \lceil S \rceil \rfloor\ (j , \mathsf{ok}\ (i , x)) & \xrightarrow[\mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ \lceil S \rceil)]{} & \mu \lfloor S \rfloor\ (i , x) \end{array}$$

  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 𝔽Fam:

$$\begin{array}{ccc} \mathit{pull} \bowtie \mathit{outl}\ ,\ N & \xrightarrow{\_\ ,\ \mathit{outr}} & f \bowtie \mathit{outl}\ ,\ \mu\ \lfloor P \otimes \lceil S \rceil \rfloor \\ {\scriptstyle \_\ ,\ \mathit{outl}}\downarrow & & \downarrow{\scriptstyle \mathit{outr}_{\bowtie}\ ,\ \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ P\ \lceil S \rceil)} \\ e \bowtie \mathit{outl}\ ,\ \mu\ \lfloor O \otimes \lceil S \rceil \rfloor & \xrightarrow[\mathit{outr}_{\bowtie}\ ,\ \mathit{forget}\ (\mathit{diffOrn\text{-}r}\ O\ \lceil S \rceil)]{} & \Sigma\ I\ (\mu\ D)\ ,\ \mu\ \lfloor S \rfloor \end{array} \tag{4.10}$$

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



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

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

where $X \boxtimes Y$, $X \boxtimes Z$, $Y \boxtimes Z$, and $(X \boxtimes Y) \boxtimes Z$ indeed give rise to products in *SliceCategory* $\mathbb{F}\textsc{am}$ $(I\ ,\ \mu\ D)$, i.e., pullbacks in $\mathbb{F}\textsc{am}$, by instantiating (4.3). What we get out of this instantiation of the lemma is a pullback in *SliceCategory* $\mathbb{F}\textsc{am}$ $(I\ ,\ \mu\ D)$ rather than $\mathbb{F}\textsc{am}$. This is easy to fix, since there is a forgetful functor from any *SliceCategory C B* to *C*:

$$
\begin{aligned}
&\textsc{SliceF} : \mathsf{Functor}\ (\mathit{SliceCategory}\ C\ B)\ C \\
&\textsc{SliceF} = \mathbf{record}\ \{\ \mathit{object} \quad = \mathsf{Slice}.T \\
&\qquad\qquad\quad ;\ \mathit{morphism} = \mathsf{SliceMorphism}.m \\
&\qquad\qquad\quad ;\ \text{proofs of laws}\ \}
\end{aligned}
$$

which is pullback-preserving. We thus get a pullback in $\mathbb{F}\textsc{am}$ of the same slices as (4.10) whose vertex is $(\mathit{pull} \bowtie \mathit{outl}\ ,\ M)$.

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

4.4 Discussion

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

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

composition).

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

Dagand and McBride [2013] provided a purely categorical treatment of ornaments using fibred category theory [Jacobs, 1999], which is quite independent of the development in this chapter, though. They established correspondences between descriptions and "polynomial functors" [Gambino and Kock, 2010] and between ornaments and "cartesian morphisms", and sketched how several operations on ornaments correspond to certain categorical notions (including pullbacks), all of which are ultimately based on the abstract notion of locally cartesian closed categories. Methodologically, there is a notable difference between their work and the categorical development in this dissertation: Dagand and McBride distinguish "software" (e.g., descriptions and ornaments) and "mathematics" (e.g., polynomial functors and cartesian morphisms) and then make a connection between them, so they can use mathematical notions as inspiration for software constructs. In the light of the propositions-as-types principle, though, a further step should be taken: rather than merely making a

connection between software artifacts that have the necessary level of detail and mathematical objects that possess desired abstract properties, we should design the software artifacts such that they satisfy the abstract properties themselves, all expressed in one uniform language, so the detail of the artifacts can be effectively managed by reasoning in terms of the abstract properties. In this spirit, category theory is regarded in this dissertation as merely providing an additional set of abstractions formulated within type theory, as opposed to being an independent formalism. (As a consequence, the reasoning is still essentially type-theoretical, rather than insisting on purely categorical arguments.) Ornaments are complex, but the complexity is necessary (as justified in **??**) and hence can only be somehow managed rather than being neglected by turning attention to similar mathematical objects. Fortunately, ornaments themselves exhibit useful categorical structure that can be expressed type-theoretically, so we are able to tame the complexity of ornaments by reasoning abstractly in terms of this categorical structure without switching to a fundamentally different formalism.

The results of this chapter are completely formalised in Agda. The setoid approach to managing morphism equivalence works reasonably well, being able to express extensional equality (e.g., in $\mathbb{F}$un and $\mathbb{F}$am), proof irrelevance (e.g., in *SpanCategory* and *SliceCategory*), and layered definitions of equality (e.g., morphism equivalence in *SquareCategory* is ultimately defined in terms of morphism equivalence in the ambient category). However, the approach works only because the formalised theory is still simple enough: for example, isomorphisms of categories in general cannot be defined sensibly, since that requires us to turn sets of objects into setoids as well, and then the setoids of morphisms have to be indexed by setoids and proved to respect equalities of the latter, which quickly becomes unmanageable. Our modelling of categories is able to evade this problem because both of the isomorphisms of categories in this dissertation (one is between $\mathbb{O}$rn and $\mathbb{F}$ref, and the other will appear in **??**) use identities as their object parts. As for formalisation of proofs, while the purely categorical lemmas can be encoded by equational reasoning combinators with a reasonable amount of effort, formalisation in general is rather difficult due to Agda's intensionality and lack of a tactic language for constructing

proofs efficiently, which together result in the need of manually constructing proof terms that require complicated handling of equalities. Also, not all the reasoning presented can be faithfully encoded: For the last two steps of the argument in Section 4.3.1, it is possible to formalise the lemma and the change of variables individually and chain them together, but the resulting isomorphisms would have a very complicated definition containing plenty of suspended *subst*s. If these isomorphisms are used to construct the refinement family in the morphism part of RSem, it would be terribly difficult to prove that the morphism part of RSem preserves equivalence. The actual formalisation circumvents the problem by fusing the lemma and the change of variables into one step to get a clean Agda definition such that RSem preserves equivalence automatically. While this kind of change of arguments for formalisation seems to be the norm (see, e.g., Avigad et al. [2007, Section 4.5]), it could occur more frequently in dependently typed programming due to the requirement that the constructed terms had better be "pretty" if they are to be referred to in later types, so even a powerful tactic language probably would not help much, since it is hard to control the form of proofs constructed by tactics. More experience in tackling this kind of proof-relevant formalisation is needed.

Bibliography

Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff [2007]. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2. doi: 10.1145/1297658.1297660. ↰ page 29

Gilles Barthe, Venanzio Capretta, and Olivier Pons [2003]. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293. doi: 10.1017/S0956796802004501. ↰ page 4

Pierre-Évariste Dagand and Conor McBride [2013]. A categorical treatment of ornaments. In *Logic in Computer Science*, LICS'13, pages 530–539. IEEE. doi: 10.1109/LICS.2013.60. ↰ page 27

Nicola Gambino and Joachim Kock [2010]. Polynomial functors and polynomial monads. arXiv:0906.4931. ↰ page 27

Robert Harper and Robert Pollack [1991]. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136. doi: 10.1016/0304-3975(90)90108-T. ↰ page 4

Bart Jacobs [1999]. *Categorical Logic and Type Theory*. Elsevier B.V. ISBN: 978-0444508539. ↰ page 27

Hsiang-Shang Ko and Jeremy Gibbons [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi: 10.2201/NiiPi.2013.10.5. ↰ page 27

Saunders Mac Lane [1998]. *Categories for the Working Mathematician*. Springer-Verlag, second edition. ISBN: 978-0387984032. ↰ pages 2 and 16

Conor McBride [1999]. *Dependently Typed Functional Programs and their Proofs*. Ph.D. thesis, University of Edinburgh. ↰ page 8

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson [2009]. Algebra of Programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579. doi: 10.1017/S0956796809007345. ↰ page 5