Bits of Category Theory

Pablo Nogueira

2nd March 2006

Contents

1 Introduction—2

2 Categories and abstraction—2

3 Direction of arrows—3

4 Definition of category—5

5 Example categories—6

6 Duality—7

7 Initial and final objects—7

8 Isomorphisms—7

9 Functors—8

10 (Co)Limits—10

(Co)Products

(Co)Products and abstraction

11 Arrow functor—15

12 Algebra of functors—16

13 Natural transformations—19

14 Representability—22

Bibliography—24

1 Introduction

The following pages are a sort of cheat-sheet where some basic notions of Category Theory are introduced with computing science in mind—or, more precisely, with functional programming in mind.

Category Theory is heavily used in programming language theory, especially in denotational semantics, algebraic specification, and program construction. The central concepts of these disciplines are usually wielded in their categorial[1] formulation because Category Theory provides a general, abstract, and uniform meta-language in which to express many ideas that have different concrete manifestations. In other words, "[C]ategory theory is not specialised to a particular setting. It is a basic conceptual and notational framework in the same sense as set theory ... though it deals with more abstract constructions." [Pie91, p.xi]

Interesting references on category theory are [SS03, Fok92, Pie91, BBvv98, BW99] and, of course, [AL88, Mac71].

2 Categories and abstraction

For mathematical structures to constitute categories one needs to identify 'entities with structure', called ***objects***, and 'structure-preserving' mappings between them, called ***arrows***. Preserving structure means preserving the property of being a valid object of the category. Due to their often graphical presentation, a collection of objects and a collection of arrows is called a ***diagram***. (We use the word 'collection' in a technical sense: a collection is an homogeneous set.)

The axioms describing what constitutes a category are rather general and wildly different mathematical structures can be 'categorised as categories'. Only arrows need satisfy minimal requirements: there must be an arrow composition operation that is partial, closed, associative, and has unique neutral element—the identity arrow, which must exist for every object.

Category Theory is *constructive* in the sense that witnesses (arrows) are always constructed in terms of compositions of other arrows rather than have their existence

[1] Following [Gol79] we use *categorial* instead of *categorical* in order to distinguish the technical from the ordinary use of the adjective.

posited. Category Theory is *coherent* in the sense that such arrows must satisfy *universal properties* (also known as *naturality* properties) expressed as equations involving universally-quantified arrows and their compositions. More precisely, many properties of diagrams do not depend on the internal structure of the particular objects under consideration and can be studied abstractly and independently of them. These universal properties are expressible *externally*, that is, purely in terms of composition of arrows.

As a contrasting illustration, Set Theory is concerned with the internal structure of sets and mappings. For instance, injective functions are characterised in terms of a property held by the elements of their domain and codomain sets:

$$\frac{f : A \to B \qquad a \in A \qquad a' \in A \qquad f(a) = f(a') \Rightarrow a = a'}{f \text{ is injective}}$$

The categorial approach abstracts away from this detail and concentrates on the external relationships between arrows. Sets considered as objects and set-theoretic total functions considered as arrows make up a category where arrow composition is function composition. The equivalent concept of injective function, namely, *monic* arrow, is defined in terms of its properties under composition:

$$\frac{f : A \to B \qquad g : C \to A \qquad h : C \to A \qquad f \circ g = f \circ h \;\Rightarrow\; g = h}{f \text{ is monic}}$$

This definition is a generalisation that applies in all categories and therefore a monic arrow in some categories may have nothing to do with the notion of injective function (Section 5).

3 Direction of arrows

Arrows will be written forwards, whether in type signatures or categorial diagrams. That is, we will write $f : A \to B$ and not $f : B \leftarrow A$. Good reasons have been given in favour of the latter style. In particular, the type of an applied function composition reads swiftly from the types of the functions involved when read from right to left—*i.e.*, in the same direction as that of the arrows—as shown in Figure 1(1). This is not the case when arrows and types are read from left to right, as shown in Figure 1(2)(3), for

composition applies its right argument first.

$$\frac{g : C \leftarrow B \qquad f : B \leftarrow A}{g \circ f : C \leftarrow A}\ (1) \qquad \frac{g : B \rightarrow C \qquad f : A \rightarrow B}{g \circ f : A \rightarrow C}\ (2)$$

$$\frac{f : A \rightarrow B \qquad g : B \rightarrow C}{g \circ f : A \rightarrow C}\ (3) \qquad \frac{f : A \rightarrow B \qquad g : B \rightarrow C}{f ; g : A \rightarrow C}\ (4)$$

Figure 1: Arrows and composition.

Furthermore, writing the target type on the left and the source type on the right is consistent with the normal notation for function application, where the arguments appear to the right of the function name, *i.e.*: $(g \circ f)\ x = g\ (f\ x)$. "[In] the alternative, so-called diagrammatical forms, one writes $x\ f$ for application and $f ; g$ for composition, where $x\ (f ; g) = (x\ f)\ g$" [BdM97, p2]. Nonetheless, there are also good reasons for writing arrows forwards:

1. It is the standard notation in mathematics and functional programming languages. It requires practice to get used to the backwards notation and we risk confusing readers unfamiliar with it. The choice is between flipping some compositions around in order to read types naturally versus flipping all arrows in order to make composition read naturally.

2. Only aesthetics or rigidity proscribes the use of a diagrammatical notation for composition alongside the normal notation for function application. There is no reason why composition cannot be used at will both in its diagrammatical or traditional form.

3. Only in the diagrammatical form does the type of composition itself read naturally, as it is only there that f is the first argument:

$$\begin{aligned} \text{Figure 1(1)}, \quad \circ \ &: \ (C \leftarrow A) \leftarrow (B \leftarrow A) \leftarrow (C \leftarrow B) \\ \text{Figure 1(2)(3)}, \quad \circ \ &: \ (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\ \text{Figure 1(4)}, \quad ; \ &: \ (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) \end{aligned}$$

   Unless, of course, one defines $g \circ f$ as the infix way of writing $\circ(f, g)$ instead of the

expected $\circ(g, f)$.

4 Definition of category

A category is identified by defining the objects, the arrows, what is composition, what is an identity arrow, and checking that categorial axioms are satisfied.

DEFINITION 4.1 A ***category*** $\mathbf{C}$ is a collection of ***objects*** $Obj(\mathbf{C})$ and a collection of arrows $Arr(\mathbf{C})$, such that:

1. For every pair of objects A and B there might be zero or more arrows from A to B. These arrows can be collected into a set which we denote by $Arr(A, B)$. Notice that $Arr(\mathbf{C})$ denotes the collection of all arrows of $\mathbf{C}$ whereas $Arr(A, B)$ denotes the collection of arrows from A to B. It is common practice to write $f : A \to B$ when $f \in Arr(A, B)$. It is also common practice to call A the *source* of f and B the *target* of f. Arrows have unique sources and targets. This is usually represented neatly in a diagram:

$$A \xrightarrow{\quad f \quad} B$$

2. There is an arrow-composition operation (denoted by ; or by $\circ$ as shown in Figure 1) with the following properties:

   (a) It is partial: two arrows f and g compose if the target of f equals the source of g.

   (b) It is closed: the resulting arrow is in $Arr(\mathbf{C})$:
   $$\frac{f : A \to B \qquad g : B \to C}{f; g \ : \ A \to C}$$

   (c) It is associative:
   $$\frac{f : A \to B \qquad g : B \to C \qquad h : C \to D}{f; (g; h) \ = \ (f; g); h}$$

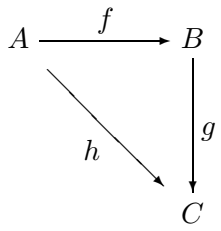
   (d) It has a left and right identity arrow for every object:
   $$\frac{f : A \to B \qquad id_A : A \to A \qquad id_B : B \to B}{id_A; f = f \qquad f; id_B = f}$$

Identities are unique:

$$u; f = f \;\Rightarrow\; u = id_A$$
$$f; u = f \;\Rightarrow\; u = id_B$$

□

Category Theory is an algebra of 'typed' arrows. By composing arrows we obtain new arrows; but arrows with the same source and target need not be equal. It is only when $h = f; g$ that we say the following diagram ***commutes***:



Universal properties are equations involving arrows expressible in terms of diagrams that commute.

5 Example categories

Categories are named after their objects. A typical category is **Set**, where objects are sets, arrows are total set-theoretic functions, and composition is function composition, which satisfies the categorial requirements for arrow composition. Another typical category is **Pre**, where objects are the members of a pre-ordered set $(A, \leq)$, arrows are pairs $(x, y) : x \to y$ such that $x{\in}A$, $y{\in}A$, $x \leq y$, and arrow composition is defined as follows:

$$\frac{(x,y) \;:\; x \to y \qquad (y,z) \;:\; y \to z}{(x,y)\;;\;(y,z) \;:\; x \to z \qquad (x,y)\;;\;(y,z) \stackrel{\text{def}}{=} (x,z)}$$

This example shows that arrows need not be functions.

A category of particular interest to functional programmers is **Type**, the category of types where objects are monomorphic types, arrows are functional programs between these types, and arrow composition is function composition.

Other interesting categories are the categories of algebras and partial algebras, where arrows are, respectively, algebra homomorphisms and partial homomorphisms.

6 Duality

For every categorial notion involving diagrams there is always a dual one in which the direction of the arrows is reversed. If $\mathbf{C}$ is a category, the *dual* or ***opposite category*** $\mathbf{C}^{\mathrm{op}}$ has the same objects and arrows as $\mathbf{C}$ only that the direction of the arrows is reversed:

$$\frac{A \in Obj(\mathbf{C})}{A \in Obj(\mathbf{C}^{\mathrm{op}})} \qquad \frac{f \in Arr(\mathbf{C}) \qquad f : A \to B}{f \in Arr(\mathbf{C}^{\mathrm{op}}) \qquad f : B \to A}$$

7 Initial and final objects

Definition 7.1 Given a category $\mathbf{C}$, $0 \in Obj(\mathbf{C})$ is an ***initial object*** iff for every object A there is a unique arrow $!_A : 0 \to A$. Accordingly, $!_0 = id_0$. Dually, given a category $\mathbf{C}$, $\mathbf{1} \in Obj(C)$ is a ***terminal object*** iff for every object A there is a unique arrow $!_A : A \to \mathbf{1}$. Accordingly, $!_{\mathbf{1}} = id_{\mathbf{1}}$. □

Arrows $x : \mathbf{1} \to A$ *from* terminal objects are called ***constants*** of A [Pie91, p17]. The motivation is that, for example, in the category of types, functional programs from the terminal type $\mathbf{1}$ (called ***unit type***) to any other type A can be put into *one-to-one* correspondence with the values in A. In other words, there is an *injective* function $i : A \to (\mathbf{1} \to A)$ such that if x is a value of type A then $i(x)$ is a value of type $\mathbf{1} \to A$, *i.e.*, a function. For instance, given the type of natural numbers:

```
data Nat = Zero | Succ Nat
```

the nullary value constructor `Zero` : `Nat` is a constant which can be lifted to a function `Zero` : $\mathbf{1} \to$ `Nat`.

8 Isomorphisms

Definition 8.1 Two objects A and B in a category are isomorphic when there are arrows $f : A \to B$ and $g : B \to A$ whose composition is the identity. In other words, $f; g = id_A$ and $g; f = id_B$. □

9 Functors

Categories are themselves mathematical structures. Functors are maps between categories which preserve the categorial structure.

DEFINITION 9.1 A ***functor*** $F : \mathbf{C} \to \mathbf{D}$ is an overloaded total map[2] between categories $\mathbf{C}$ and $\mathbf{D}$ mapping objects to objects and arrows to arrows while preserving composition and identities. More precisely,

$$F : Obj(\mathbf{C}) \to Obj(\mathbf{D})$$

$$F : Arr(\mathbf{C}) \to Arr(\mathbf{D})$$

such that:

1. $\forall A \in Obj(\mathbf{C}).\ F(A) \in Obj(\mathbf{D})$

2. If the functor is ***covariant*** in its arrow argument then:

$$\frac{f \in Arr(\mathbf{C}) \qquad f\ :\ A \to B}{F(f) \in Arr(\mathbf{D}) \qquad F(f) : F(A) \to F(B)}$$

3. If the functor is ***contravariant*** in its arrow argument then:

$$\frac{f \in Arr(\mathbf{C}) \qquad f\ :\ B \to A}{F(f) \in Arr(\mathbf{D}) \qquad F(f) : F(A) \to F(B)}$$

   However, a functor is just 'contravariant' when $f : A \to B$ but $F(f) : F(B) \to F(A)$.

4. Finally, and more importantly, the functor preserves the categorial structure:

$$\frac{f \in Arr(\mathbf{C}) \qquad g \in Arr(\mathbf{C})}{F(f;g)\ =\ F(f)\ ;\ F(g)} \qquad\qquad \frac{A \in Obj(\mathbf{C})}{F(id_A)\ =\ id_{F(A)}}$$

□

Figure 2 characterises a functor diagrammatically. If F is a functor and the diagram in $\mathbf{C}$ commutes, the diagram in $\mathbf{D}$ also commutes.

[2] Or if the reader prefers, two maps with overloaded name.

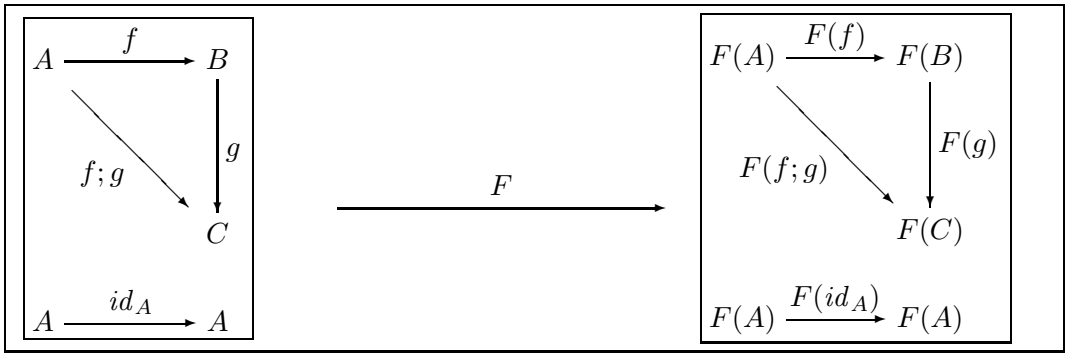


Figure 2: A functor F diagrammatically.

For simplicity, Definition 9.1 defines a unary functor. Functors of any arity are defined in terms of cartesian products of categories.

DEFINITION 9.2 The ***product category*** of two categories $\mathbf{C}$ and $\mathbf{D}$, denoted $\mathbf{C} \times \mathbf{D}$, is a category where:

$$\begin{aligned} Obj(\mathbf{C} \times \mathbf{D}) &\stackrel{\text{def}}{=} Obj(\mathbf{C}) \times Obj(\mathbf{D}) \\ Arr(\mathbf{C} \times \mathbf{D}) &\stackrel{\text{def}}{=} Arr(\mathbf{C}) \times Arr(\mathbf{D}) \end{aligned}$$

such that:

$$\frac{f \in Arr(\mathbf{C}) \qquad f : A \to C \qquad g \in Arr(\mathbf{D}) \qquad g : B \to D}{(f,g) \in Arr(\mathbf{C} \times \mathbf{D}) \qquad (f,g) : (A,B) \to (C,D)}$$

□

The previous definition can be trivially extended to n-tuples; we talk then about n-product categories. For instance, a *binary functor* (or *bifunctor*) is a functor from a product category to another category, *e.g.*, $F : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$, and similarly for n-functors. We will be mostly interested in *endofunctors*, that is, in functors from $\mathbf{C}^n$ to $\mathbf{C}$, where $\mathbf{C}^n$ is the n-product of $\mathbf{C}$. For the sake of clarity, let us illustrate how Definition 9.1 is adapted for a binary covariant functor $F : \mathbf{C}^2 \to \mathbf{C}$:

$$\frac{(A,B) \in Obj(\mathbf{C}^2)}{F(A,B) \in Obj(\mathbf{C})} \qquad \frac{(f,g) \in Arr(\mathbf{C}^2)}{F(f,g) \in Arr(\mathbf{C})} \qquad \frac{(f,g) : (A,B) \to (C,D)}{F(f,g) : F(A,B) \to F(C,D)}$$

In the category of types, a functor F : **Type** $\rightarrow$ **Type** at the object level is a type operator that maps types to types: if F is a type operator and A is a manifest type then $F(A)$ is a manifest type. At the arrow level, *i.e.*, functional programs, F must satisfy the following:

$$\begin{aligned} F(f) &: F(A) \rightarrow F(B) \\ F(f;g) &= F(f);F(g) \\ F(id_A) &= id_{F(A)} \end{aligned}$$

Which means that at the arrow level F is the **`map`** function for the type operator. For example, in the case of type operator `List`:

```
map f :: List a → List b
map (f ‘;‘ g)       ==  map f ‘;‘ map g
map (id :: a → a)   ==  (id :: List a → List a)
```

where `f ‘;‘ g` = `g` ∘ `f`. We have used explicit type annotations to illustrate the types of each **`id`** instance.

Sections 10.1 and 11 present examples of binary functors.

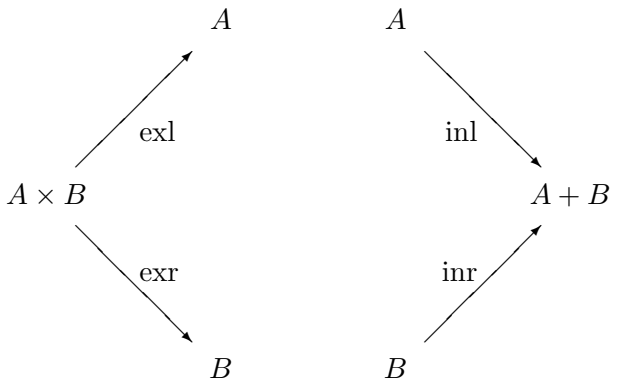
10 (Co)Limits

A limit is a solution to a diagram, *i.e.*, another diagram consisting of an object and a collection of arrows that satisfies the universal property that any other solution factors uniquely, *i.e.*, there is a unique arrow from the other solution to the object in the limit diagram which makes the combined diagrams commute. The colimit is the solution in the dual diagram. Limits and colimits can be studied bottom-up from empty diagrams by adding objects and arrows. In the next sections we only present one example.

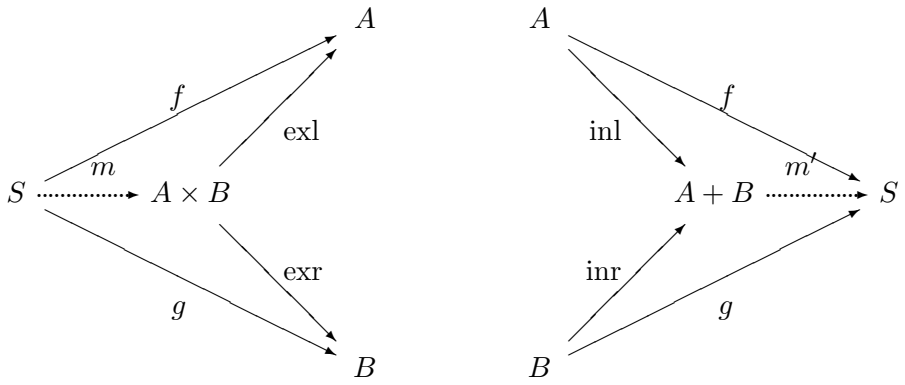
10.1 (Co)Products

A (co)product is the (co)limit of a diagram involving two objects A and B and no arrows. Following [SS03] we present both notions simultaneously. The product is an object $A \times B$ and two arrows exl and exr. The coproduct is an object $A + B$ and two

arrows inl and inr:



such that for any other similar solution (object S with arrows f and g), there is a unique mediating arrow from it to the product (or from the coproduct to it) that makes the following diagrams commute:



That is:

$$m; \mathrm{exl} = f \ \wedge \ m; \mathrm{exr} = g$$

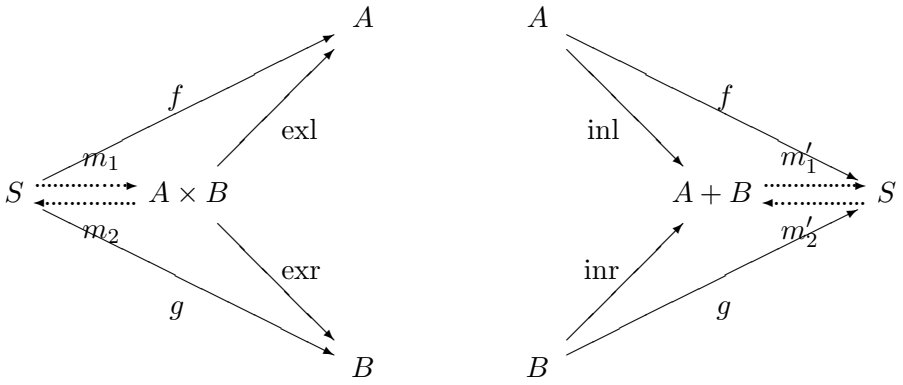
$$\mathrm{inl}; m' = f \ \wedge \ \mathrm{inr}; m' = g$$

Both m and m' are unique for every solution diagram, *i.e.*, they are uniquely determined from the arrows involved. Following [Fok92, MFP91] we make this functional relationship explicit and use functions $\triangle$ and $\triangledown$ such that:

$$m \stackrel{\text{def}}{=} f \triangledown g$$

$$m' \stackrel{\text{def}}{=} f \triangle g$$

An important consequence of universality is that any other diagram solution is isomorphic. If S is another product there is a unique arrow $m_2 : A \times B \to S$. If S is another coproduct there is a unique arrow $m'_2 : S \to A + B$. Because mediating arrows are unique and the resulting diagrams commute, the composition of mediating arrows is the identity and their source and targets are isomorphic (Section 8). More precisely:



$$\left.\begin{array}{rcl} m_1;\mathrm{exl} & = & f \\ m_2; f & = & \mathrm{exl} \end{array}\right\} \Rightarrow m_1; m_2; f = f \qquad f; m'_2; m'_1 = f \Leftarrow \left\{\begin{array}{rcl} \mathrm{inl}; m'_1 & = & f \\ f; m'_2 & = & \mathrm{inl} \end{array}\right.$$

$$\left.\begin{array}{rcl} m_1;\mathrm{exr} & = & g \\ m_2; g & = & \mathrm{exr} \end{array}\right\} \Rightarrow m_1; m_2; g = g \qquad g; m'_2; m'_1 = g \Leftarrow \left\{\begin{array}{rcl} \mathrm{inr}; m'_1 & = & g \\ g; m'_2 & = & \mathrm{inr} \end{array}\right.$$

Consequently:

$$m_1; m_2 = id_S \wedge m_2; m_1 = id_{A\times B} \qquad m'_2; m'_1 = id_S \wedge m'_1; m'_2 = id_{A+B}$$

10.2 (Co)Products and abstraction

The definition of product captures the general notion of an object that is uniquely formed by combining two objects such that we can recover them via arrows exl and exr *irrespective of the internal structure* of that composite object. For example, in the category of sets, the categorial product is the cartesian product, which can be defined internally in many ways:

$$A \times B \stackrel{\mathrm{def}}{=} \{\ \{\{a\},\{a,b\}\} \mid a{\in}A \wedge b{\in}B\ \}$$
$$A \times B \stackrel{\mathrm{def}}{=} \{\ \{\{b\},\{a,b\}\} \mid a{\in}A \wedge b{\in}B\ \}$$
$$A \times B \stackrel{\mathrm{def}}{=} \{\ \{\{a,0\},\{b,1\}\} \mid a{\in}A \wedge b{\in}B\ \}$$

The product object is a generalisation, *i.e.*, an *abstract set* with operations for construction and observation. In the category of types, $A \times B$ is an *abstract type* (a composite of A and B with two selector operators) which abstracts from the internal structure (representation) of the object.

A coproduct is a type into which we can inject two types using the two arrows. The mediating arrow $f \vartriangle g$ provides lifted construction in products and $f \triangledown g$ provides lifted discrimination plus selection in coproducts as shown in Figure 3, where lifting refers to the process of turning values into functions.

```
exl  :: Prod a b → a
exr  :: Prod a b → b
△ :: (c → a) → (c → b) → (c → Prod a b)

(f △ g) x = prod (f x) (g x)
prod :: a → b → Prod a b

inl  :: a → CoProd a b
inr  :: b → CoProd a b
▽ :: (a → c) → (b → c) → (CoProd a b → c)

(f ▽ g) x = if isl x then (f ∘ asl) x else (f ∘ asr) x
asl  :: CoProd a b → a
asr  :: CoProd a b → b
isl  :: CoProd a b → Bool
```

Figure 3: Type `Prod` stands for a product type and `CoProd` for a coproduct type.

Notice that types `Prod` and `CoProd` are abstract, we have not provided their definition in terms of concrete types. In a functional language (*i.e.*, Haskell), binary products and coproducts are manipulated through concrete representations introduced in type definitions, *i.e.*, cartesian products and disjoint sums (Figure 4).

Binary (co)products are generalised to n-ary (co)products trivially: the (co)product is an object and n arrows [Pie91].

Interestingly, product and coproduct construction, *i.e.*, $\times$ and $+$, are both endofunctors from $\mathbf{C}^2$ to $\mathbf{C}$. It is standard to deviate from the prefix functor application notation

```
type Prod   a b = (a,b)
data CoProd a b = Inl a | Inr b

exl = fst
exr = snd
inl = Inl
inr = Inr
isl (Inl _) = true
isl (Inr _) = false
asl (Inl x) = x
asl (Inr _) = undefined
asr (Inl _) = undefined
asr (Inr y) = y
```

Figure 4: An 'implementation' of Figure 3 which describes the internal structure of the objects and arrows.

and write $A \times B$ instead of $\times(A, B)$ and similarly for $+$. More precisely:

$$\frac{(A,B) \in Obj(\mathbf{C}^2)}{A \times B \in Obj(\mathbf{C})} \qquad \frac{(f,g) \in Arr(\mathbf{C}^2)}{f \times g \in Arr(\mathbf{C})} \qquad \frac{f : A \to C \qquad g : B \to D}{f \times g : A \times B \to C \times D}$$

In the category of types, at the arrow level $\times$ corresponds to the function:

```
map_Prod :: (a → c) → (b → d) → Prod a b → Prod c d
map_Prod f g = (f ∘ exl) △ (g ∘ exr)
```

In the case of concrete tuple types (Figure 4), the function can be written more familiarly:

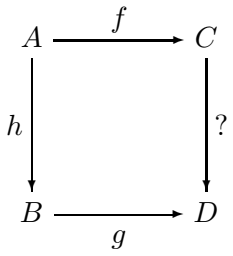
```
map× :: (a → c) → (b → d) → Prod a b → Prod c d
map× f g (x,y) = (f x, g y)
```

Similarly, for coproducts:

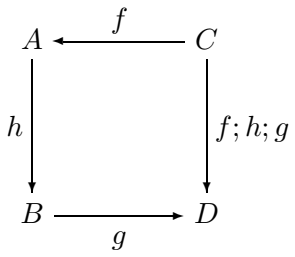
```
map_CoProd :: (a → c) → (b → d) → CoProd a b → CoProd c d
map_CoProd f g = (inl ∘ f) ▽ (inr ∘ g)

map+ :: (a → c) → (b → d) → CoProd a b → CoProd c d
map+ f g (Inl x) = Inl (f x)
map+ f g (Inr y) = Inr (g y)
```

11 Arrow functor

In Section 4 we introduced the notation $Arr(A, B)$ to express the collection of arrows from A to B in a given category $\mathbf{C}$. Interestingly, Arr can be understood as an endofunctor from $\mathbf{C}^2$ to $\mathbf{C}$. In the category of types, for any $(A, B) \in Obj(\mathbf{Type}^2)$, $Arr(A, B)$ is another type: the type of functions (arrows in $\mathbf{Type}$) from A to B. This functor has the peculiar characteristic that it is contravariant on its first argument. Why this is so becomes apparent by looking at the following diagram:



At the arrow level, we cannot define $Arr(f, g) : Arr(A, B) \to Arr(C, D)$ by composing f and g with arrows $h \in Arr(A, B)$ to yield an arrow in $Arr(C, D)$. We can do it if Arr is contravariant on its first argument:



DEFINITION 11.1 Given a category $\mathbf{C}$, the ***arrow functor*** $Arr : \mathbf{C}^2 \to \mathbf{C}$ is defined as follows:

$$\frac{(A, B) \in Obj(\mathbf{C^2})}{Arr(A, B) \in Obj(\mathbf{C})}$$

$$\frac{f : C \to A \qquad g : B \to D \qquad h \in Arr(A, B)}{Arr(f, g) \ : \ Arr(A, B) \to Arr(C, D) \qquad (Arr(f, g))(h) \stackrel{\text{def}}{=} f; h; g}$$

□

In the category of types, $Arr(A, B)$ is the function space $A \to B$. At the arrow level, $f \to g$ can be written using more familiar Haskell notation:

```
map→ :: (c → a) → (b → d) → (a → b) → (c → d)
map→ f g h = g ∘ h ∘ f
```

12 Algebra of functors

Just like there is an algebra of manifest types and type operators which can be combined to form type-terms, there is an algebra of objects and functors which can be combined to form object expressions. The following definitions provide the machinery.

Identity functor: The identity functor $Id : \mathbf{C} \to \mathbf{C}$ is defined as follows:

$$\frac{A \in Obj(\mathbf{C})}{Id(A) \in Obj(\mathbf{C}) \qquad Id(A) \stackrel{\text{def}}{=} A} \qquad \frac{f \in Arr(\mathbf{C}) \qquad f : A \to B}{Id(f) \in Arr(\mathbf{C}) \qquad Id(f) \stackrel{\text{def}}{=} f}$$

Constant functor: The constant functor $K_B : \mathbf{C} \to \mathbf{C}$ for *every* object B of $\mathbf{C}$ is defined as follows:

$$\frac{A \in Obj(\mathbf{C})}{K_B(A) \in Obj(\mathbf{C}) \qquad K_B(A) \stackrel{\text{def}}{=} B} \qquad \frac{f \in Arr(\mathbf{C}) \qquad f : C \to D}{K_B(f) \in Arr(\mathbf{C}) \qquad K_B(f) \stackrel{\text{def}}{=} id_B}$$

Polynomial functors and pointwise lifting: Objects described by object expressions can be obtained from applications of the constant functor, the identity functor, and the product and coproduct functors to other objects. For example, if A, B and C

are objects of $\mathbf{C}$, so is $A + (B \times C)$. The object is not named but written in terms of applications of functors to objects.

Functors can also be defined in terms of object expressions. In the previous expression if A stands for a free variable instead of an object, the object turns into a functor $(\cdot + (B \times C)) : \mathbf{C} \to \mathbf{C}$, where we indicate by $\cdot$ the place were the actual parameter would go. More commonly, functors are named in definitions:

$$F(X) \stackrel{\text{def}}{=} X + (B \times C)$$

(Inexplicably, Lambda Calculus notation has never caught on in Category Theory or in maths as a whole.)

It is sometimes convenient to define F only in terms of the functors involved and not in terms of functors and objects. To do that we define a notion of ***pointwise lifting*** for functors.

DEFINITION 12.1 Let $F : \mathbf{C}^2 \to \mathbf{C}$ be a functor. The lifting of F, denoted $\dot{F}$ is defined as follows:

$$\frac{A \in Obj(\mathbf{C}) \qquad G : \mathbf{C} \to \mathbf{C} \qquad H : \mathbf{C} \to \mathbf{C}}{\dot{F}(G,H) : \mathbf{C} \to \mathbf{C} \qquad (\dot{F}(G,H))(A) \stackrel{\text{def}}{=} F(G(A), H(A))}$$

□

With this definition at hand it is not difficult to check that $F(X) \stackrel{\text{def}}{=} X + (B \times C)$ can be defined in terms of functors and pointwise-lifted functors:

$$F \stackrel{\text{def}}{=} Id \dot{+} (K_B \dot{\times} K_C)$$

At the object level:

$$\begin{aligned} F(A) &= (Id \dot{+} (K_B \dot{\times} K_C))(A) \\ &= Id(A) + (K_B \dot{\times} K_C)(A) \\ &= A + (K_B(A) \times K_C(A)) \\ &= A + (B \times C) \end{aligned}$$

At the arrow level:

$$\begin{aligned} F(f) &= (Id \dot{+} (K_B \dot{\times} K_C))(f) \\ &= Id(f) + (K_B \dot{\times} K_C)(f) \\ &= f + (K_B(f) \times K_C(f)) \\ &= f + (id_B \times id_C) \end{aligned}$$

The last expression is more commonly written in Haskell as follows for fixed types `C` and `B`:

```
map+ f (map× (id :: B → B) (id :: C → C))
```

Pointwise lifting produces ***higher-order functors***:

$$\frac{F : \mathbf{C}^2 \to \mathbf{C}}{\dot{F} : \mathbf{Func}(\mathbf{C}, \mathbf{C})^2 \to \mathbf{Func}(\mathbf{C}, \mathbf{C})}$$

where $\mathbf{Func}(\mathbf{C}, \mathbf{C})$ is the category of functors from $\mathbf{C}$ to $\mathbf{C}$ (yep, functors make up a category, see Section 13). It is common to write $\mathbf{Func}(\mathbf{C}, \mathbf{C})$ as $\mathbf{C} \to \mathbf{C}$, making the 'type signature' of $\dot{F}$ more obvious to a functional programmer:

$$\dot{F} : (\mathbf{C} \to \mathbf{C}) \to (\mathbf{C} \to \mathbf{C})$$

The final touch is provided by lifting objects to functors. An object $A \in Obj(\mathbf{C})$ is lifted to a functor $\dot{A} : \mathbf{1} \to \mathbf{C}$, where $\mathbf{1}$ is the initial category (the initial object in the category of *small* categories, see Section 13). If we drop the notational distinction between lifted and regular functors, objects, and lifted products, we end up in a 'language-game' similar to that of values and (higher-order) functions, or manifest types and (higher-order) type operators. This facilitates the treatment of type operators as functors.

The last ingredient in this setting is the introduction of fixed points to account for recursive equations involving functors from $\mathbf{C}^n$ to $\mathbf{C}$. Let us state here the definition for $n = 1$:

Definition 12.2 Let $\mathbf{C}$ be a category and $F : \mathbf{C} \to \mathbf{C}$ a functor. A ***fixed point*** of F is a pair (A, α) where $A \in Obj(\mathbf{C})$ and $\alpha : F(A) \to A$ is an isomorphism. □

The technical machinery needed to explain the definition in detail is beyond the scope

of this presentation. Let us just mention that the fixed points of F form a category and the least fixed point is the initial object. Such category is a subcategory of the category of F-algebras [Pie02]. The proof of existence of the initial algebra is basically a categorial generalisation of Tarski's fixed-point theorem [Pie91, p61–72].

13 Natural transformations

Categories are themselves mathematical structures and can be taken to be objects of another category where functors are the arrows. To avoid circular notions such as the category of all categories (whose objects are all categories) that may lead to paradoxes similar to Russell's in Set Theory [Ham82], categories are classified into ***small*** and ***large***, where in the former objects are not categories.

It is interesting to consider whether *arrows* of a category are *objects* of another category and what would then be the corresponding notion of arrow in this second category, called an ***arrow category***. Functors are maps between categories which preserve the categorial structure. A ***functor category*** is an example of an arrow category: objects are functors between small categories; arrows are called natural transformations.

Definition 13.1 Let $\mathbf{C}$ and $\mathbf{D}$ be categories. Let $\mathbf{Func}(\mathbf{C}, \mathbf{D})$ be the category of functors from $\mathbf{C}$ to $\mathbf{D}$ and F and G two functors (objects) of this category that have the same variance. A ***natural transformation*** $\eta : F \dot{\rightarrow} G$ is an arrow in $\mathbf{Func}(\mathbf{C}, \mathbf{D})$. (The notation $\dot{\rightarrow}$ is introduced for arrows between functors.) More precisely, η is a family of $\mathbf{D}$-arrows indexed by objects of $\mathbf{C}$:

$$\eta \;=\; \{\; \eta_X \in Arr(F(X), G(X)) \mid X \in Obj(\mathbf{C}) \;\}$$

In words, a natural transformation $\eta : F \dot{\rightarrow} G$ assigns to each object $A \in Obj(\mathbf{C})$ an arrow $\eta_A \in Arr(F(A), G(A))$. The arrows must satisfy the following coherence (or naturality) property: depending on whether F and G are both covariant or contravariant,

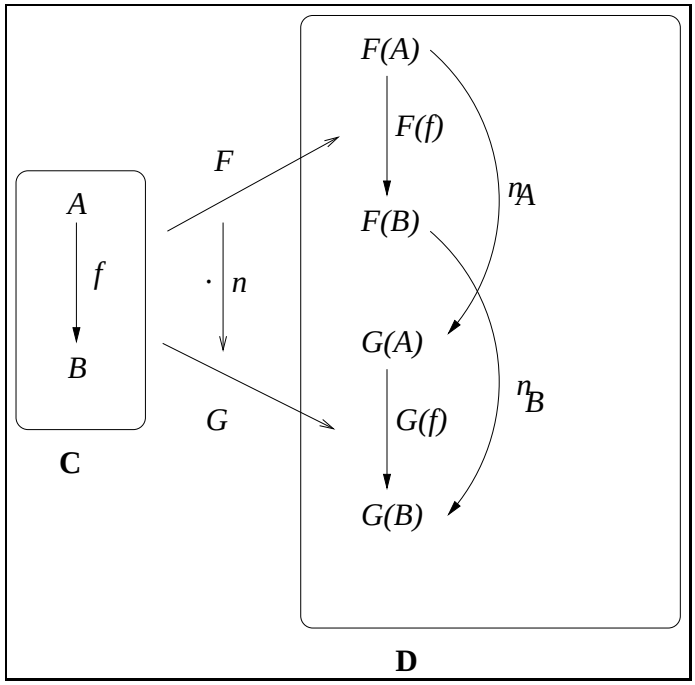
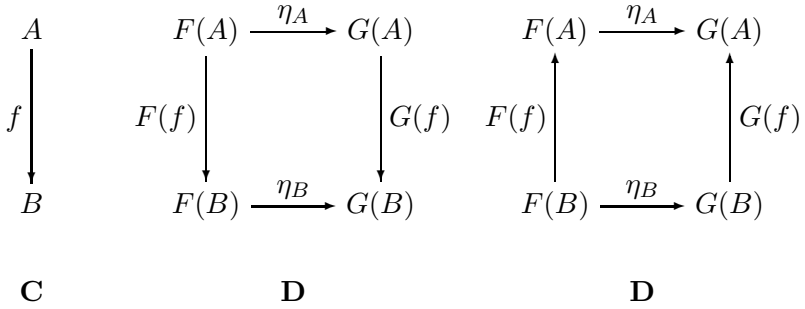


Figure 5: η is a natural transformation when $\eta_A; G(f) = F(f); \eta_B$ for every pair of objects A and B in $\mathbf{C}$.

the squared diagrams in $\mathbf{D}$ commute respectively:



□

Figure 5 is a more illustrative depiction of the definition when F and G are both covariant. Given $A \in Obj(\mathbf{C})$, we can draw an arrow η_A from $F(A)$ to $G(A)$, *i.e.*, between two objects in $Obj(\mathbf{D})$ arising from the same object A by two different functors. And this can be done for *any* object in $Obj(\mathbf{C})$. Furthermore, for any other object $B \in Obj(\mathbf{C})$, there are two ways of defining an arrow from $F(A)$ to $G(B)$, namely,

$F(f);\eta_B$ and $\eta_A;G(f)$. Both must be equal:

$$F(f)\ ;\ \eta_B\ =\ \eta_A\ ;\ G(f)$$

In the category of types, natural transformations are polymorphic functions between type operators. For example:

```
flatten :: List (List a) → List a
flatten xs = foldr (++) []
```

is a natural transformation:

```
flatten  : List;List ·→ List
```

as proven by the following equation:

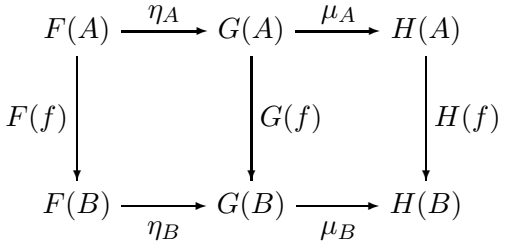
```
(map ';' map) f ';' flatten_b ==  flatten_a ';' map f
```

where `flatten`$_a$ and `flatten`$_b$ are instances of `flatten` at any two types `a` and `b` respectively. The equation can be put in the general form as follows:

$F \stackrel{\text{def}}{=}$ `List;List`, which at the arrow level is **`map ';' map`**.
$G \stackrel{\text{def}}{=}$ `List`, which at the arrow level is **`map`**.
$\eta_A \stackrel{\text{def}}{=}$ `flatten`$_a$
$\eta_B \stackrel{\text{def}}{=}$ `flatten`$_b$

Both sides of the equation are functions of type `List (List a)` → `List b` which are equal. The first function maps `f :: a` → `b` over the list of list of `a`s and then flattens the resulting list of `b`s. The second function flattens the list of lists of `a`s into a list of `a`s and then maps `f` to get a list of `b`s.

That functors as objects and natural transformations as arrows make up a category is illustrated by the following diagram. The composition of natural transformations is associative: given $\eta : F \dot{\to} G$ and $\mu : G \dot{\to} H$, their composition $\eta;\mu : F \dot{\to} H$ defined by $(\eta;\mu)_A = \eta_A;\mu_A$ for every $A \in Obj(C)$ is natural (*i.e.*, the diagram commutes):



The identity natural transformation $\iota : F \stackrel{\cdot}{\to} F$ is the collection of identities of the objects in the image of F, *i.e.*, $\iota_A = id_{F(A)}$.

14 Representability

The idea of representability is to state the conditions under which a functor $F : \mathbf{C} \to \mathbf{Set}$ can be identified uniquely with an object of $\mathbf{C}$.

An important ingredient in stating the conditions is the notion of ***hom-functor*** which enables us to talk about an object by means of the collection of arrows with that object as source.[3] Recall from Section 11 that $Arr : \mathbf{C}^2 \to \mathbf{C}$ is a binary functor. However, every object A in $\mathbf{C}$ determines a collection of arrows $Arr(A, \cdot)$, *i.e.*, the collection of arrows with A as source:

$$Arr(A, \cdot) \stackrel{\text{def}}{=} \{\ f \in Arr(\mathbf{C}) \mid f : A \to X \ \wedge \ X \in Obj(\mathbf{C})\ \}$$

In Section 2 we have defined 'collection' as 'homogeneous set', which means $Arr(A, \cdot)$ is an object in **Set**. (Actually, in order to avoid paradoxes we must make sure that collections of arrows are sets. We impose this as a condition and call categories that satisfy it ***locally small***.) Consequently, for every object A there is an hom-functor $Arr(A, \cdot)$. To be consistent with our notation for functors, in the following definition we write Hom_A instead of $Arr(A, \cdot)$.

Definition 14.1 (hom-functor) Let $\mathbf{C}$ be a locally small category and **Set** the category of sets. Every object $A \in Obj(\mathbf{C})$ determines a ***hom-functor*** $\mathrm{Hom}_A : \mathbf{C} \to \mathbf{Set}$ defined as follows:

$$\frac{X \in Obj(\mathbf{C})}{\mathrm{Hom}_A(X) \in Obj(\mathbf{Set}) \qquad \mathrm{Hom}_A(X) \stackrel{\text{def}}{=} Arr(A, X)}$$

$$\frac{f \in Arr(\mathbf{C})}{\mathrm{Hom}_A(f) \in Arr(\mathbf{Set})}$$

$$\frac{f : X \to Y}{\mathrm{Hom}_A(f) : \mathrm{Hom}_A(X) \to \mathrm{Hom}_A(Y) \qquad (\mathrm{Hom}_A(f))(g) \stackrel{\text{def}}{=} g; f}$$

[3] The name 'hom-functor' comes from the fact that arrows are called *homomorphisms* by some authors. We have already used 'arrow-functor' with a different meaning in Section 11.

□

In words, at the object level Hom_A maps an object X of $\mathbf{C}$ to the set of arrows of $\mathbf{C}$ with A as source and X as target. At the arrow level, Hom_A maps an arrow $f : X \to Y$ to a function from the set $\mathrm{Hom}_A(X)$ to the set $\mathrm{Hom}_A(Y)$, that is, a function that given $g \in \mathrm{Hom}_A(X)$, *i.e.*, an arrow from A to X, returns $g; f \in \mathrm{Hom}_A(Y)$, *i.e.*, an arrow from A to Y.

Definition 14.2 (representable functor) Let $\mathbf{C}$ be a locally small category, a functor $F : \mathbf{C} \to \mathbf{Set}$ is represented by an object $A \in Obj(\mathbf{C})$ iff F and Hom_A are isomorphic objects in $\mathbf{Func}(\mathbf{C}, \mathbf{Set})$, *i.e.*, there are two arrows η and μ—natural transformations in $\mathbf{Func}(\mathbf{C}, \mathbf{Set})$—such that:

$$\begin{aligned} \eta &: F \dot{\to} \mathrm{Hom}_A \\ \mu &: \mathrm{Hom}_A \dot{\to} F \\ \eta; \mu &= \iota_F \\ \mu; \eta &= \iota_{\mathrm{Hom}_A} \end{aligned}$$

□

where ι is the identity natural-transformation.

References

[AL88] Andrea Asperti and Giuseppe Longo. Categories, types, and structures: an introduction to category theory for the working computer scientist. Electronic book, 1988.

[BBvv98] Roland Backhouse, Marcel Bijsterveld, Rik van Geldrop, and Jaap van der Woude. Category theory as coherently constructive lattice theory. Working document, 12 June 1998.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice-Hall, 1997.

[BW99] Michael Barr and Charles Wells. Category theory. Lecture Notes, ESSLLI, 1999.

[Fok92] Maarten M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1992.

[Gol79] Robert Goldblatt. *Topoi: The Categorial Analysis of Logic.* North-Holland, New York, 1979.

[Ham82] A. G. Hamilton. *Numbers, Sets and Axioms: The Apparatus of Mathematics.* Cambridge University Press, 1982.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS.* Springer-Verlag, 1991.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[SS03] Harold Simmons and Andrea Schalk. *Category Theory in Four Easy Movements*. Online Book, University of Manchester, October 2003.