Category Theory

Yu-Yang Lin
1228863
MSci in Computer Science

Individual Study 2 Report
Semester 1
Supervised by Dr. Paul B. Levy

School of Computer Science
University of Birmingham
United Kingdom
10th December, 2015

To my family and friends

Acknowledgements

I want to thank Paul for teaching us about Category Theory and supervising us, as well as my fellow Category Theorists, for being in the meetings–helping me understand many of the topics. Those were very interesting meetings.

I would also like to thank Dan Ghica, who suggested I should take Individual Study on Category Theory with Paul.

Finally, I would like to thank Martín Escardó for helping me with Agda proofs.

Contents

Chapter 1

What is a Category?

1.1 Definition

A **category** consists of the following:

- A **class** of objects obC
  if obC is a set, this is a *small* category
- For any objects $A, B \in C$
  a set $hom_C(A,B)$ of morphisms $f : A \to B$
  i.e. a homset from A to B
- For each object A
  a morphism $id_A : A \to A$
  i.e. an identity morphism for every object
- For any objects A, B, C and morphisms $f : A \to B$ and $g : B \to C$
  the **composition** is $f;_{A,B,C} g : A \to C$
  (also written $g \circ f$, the subscripts for (;) are required but sometimes shorthanded)
  such that:
  - For any objects A, B and $f : A \to B$
    $id_A; f = f = f; id_B$ [i.e. composing the identity with f equals f]
  - For any objects A, B, C, D and morphisms $f : A \to B$ and $g : B \to C$ and $h : C \to D$
    $f;(g;h) = (f;g);h$ [i.e. composition is associative]

1.2 Examples

1.2.1 The category of sets, Set

- An object is a set

- A morphism $A \to B$ is a function, the homset contains all function definitions as well as undefinable ones
- The identity id_A is the identity function $x \mapsto x$
- The composite $A \xrightarrow{f} B \xrightarrow{g} C$ is the composition of a function $[x \mapsto g(f(x))]$

1.2.2 The category of matrices, Mat

- An object is a natural number $n \in \mathbb{N}$
- The morphisms $m \to n \in hom(m, n)$ are all $m \times n$ matrices (m rows, n cols). i.e. every morphism is a collection of field elements (t_i^j) where i runs from 0 to $m-1$ and j from 0 to $n-1$.
- The identity id_n is the identity matrix of $n \times n$ (morphism $n \to n$) such that it acts as the identity for matrix multiplication
- The composite of $(s_i^j) : m \to n \in hom(m, n)$ and $(t_j^k) : n \to p \in hom(p, m)$ is matrix multiplication where the product $(s_i^j; t_j^k) : m \to p \in hom(p, m)$ is

  $(s_i^j \mid {}^{i<m}_{j<n})\ (t_j^k \mid {}^{j<n}_{k<p}) = (\sum_{j<n} s_i^j t_j^k \mid {}^{i<m}_{k<p})$

1.2.3 The category of towns in Britain

- An object is a town in Britain
- The morphisms $A \to B \in hom(A, B)$ are all routes from A to B. i.e. a finite sequence (list) of adjacent towns.
- The identity id_A is a route from a town A to itself which consists of of a single element A.

  i.e. the identity always is a list of length 1.
- The composite of $f : A \to B$ and $g : B \to A$ is the concatenations of the routes f and g.

  e.g. if $f = [A, A_1, A_2, A_3, B]$ and $g = [B, B_1, B_2, B_3, C]$ then $f; g = [A, A_1, A_2, A_3, B, B_1, B_2, B_3, C]$

1.3 Homework Examples

1.3.1 The category of groups and group homomorphisms, Grp

- An object is a group
- The morphism $h : G \to H \in hom(G, H)$ is a group homomorphism.

  i.e. a function that preserves the algebraic structure from group $(G, *)$ in group $(H, \bullet)$:

given $a * b = c$ we have $h(a) \bullet h(b) = h(c)$

i.e. $h(a * b) = h(a) \bullet h(b)$

- The identity id_G is a homomorphism that maps G to itself.

  More specifically, given $G = (X, e, *)$, $x \in X$:

  $id_G : G \to G$

  $id_G = x \mapsto x$

- The composite of morphisms $h : G \to H$ and $k : H \to K$ is $h;k : G \to K$ or $k \circ h : G \to K$

  i.e. a homomorphism from G to K

A **group** (G) is a set with an operation $(\bullet : G^2 \to G)$ that combines two elements in G to form a third element in G. In computer science, we generally include element e to remove existentials. For $(G, \bullet)$ to be a group, it must satisfy:

- **Closure** : $(a, b \in G) \Rightarrow (a \bullet b \in G)$.
- **Associativity** : $\forall a, b, c \in G \,.\, (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- **Identity** : $\forall a \in G \,.\, \exists e \in G \,.\, e \bullet a = a \bullet e = a$
- **Inverse Element** : $\forall a \in G \,.\, \exists b \in G \,.\, a \bullet b = b \bullet a = e$

  $*$ note: a group without inverse functions is a **monoid**.

  i.e. for morphism $\mathbb{N} \to \mathbb{N}$, the set of all functions is a monoid while the set of all bijections is a group.

1.3.2 The category of vector spaces and linear transformations, K-Vect

- An object is a vector space (a scalable collection of vectors) over a fixed field K
- The morphism $V \to W \in hom(V, W)$ is a **K-linear map**, transformations between two linear subspaces that preserve addition and scalar multiplication
- The identity id_V is an endomorphism on V (a morphism from vector space V to itself)

  More specifically, given $v \in V$:

  $id_V = v \mapsto v$

- The composite of morphisms $f : V \to W$ and $g : W \to Y$ is $f;g : V \to Y$

1.3.3 The category of posets (partially ordered sets) and monotone functions

- An object is a poset

- For $(S, \leq)$ and $(T, \leq)$, the morphism $f : S \to T \in hom(S, T)$ is a **order-preserving** or **monotone** function.

  A monotone function is one such that:

  $\forall x, y \in S.(x \leq y) \Rightarrow f(x) \leq f(y)$

- The identity id_S is a morphism from S to itself
- The composite of $f : S \to T$ and $g : T \to U$ is $f; g : S \to U$ or $(g \circ f) : S \to U$, which is also monotone

Posets $(S, \leq)$ formalise the intuitive concept of ordering, sequencing, or arrangement of the elements of a set.

Posets consist of a set (S) and a binary relation ($\leq$) that indicates a partial order.

A **partial order** is a binary relation between elements of a set ($a, b, c \in S$) that is:

- **reflexive**: $a \leq a$
- **antisymmetric**: $(a \leq b) \wedge (b \leq a) \Rightarrow a = b$
- **transitive**: $(a \leq b) \wedge (b \leq c) \Rightarrow a \leq c$

1.3.4 The category of set relations, Rel

- An object is a set
- The morphism $f : A \to B \in hom(A, B)$ is a relation.

  A relation is between A and B is defined by the crossproduct of both sets, i.e. a set of tuples.

  $(R \subseteq A \times B)$ and $\{(a, b) \mid (a, b) \in R \wedge a \in A \wedge b \in B\}$
- The identity morphism $id_A : A \to A$ is the identity relation $\{(a, a) \in A^2 \mid a \in A\}$
- The composite of $R : A \to B$ and $S : B \to C$ is $R; S : A \to C$ given by:
  $S \circ R = \{(a, c) \in A \times C \mid \exists b \in B.(a, b) \in R \wedge (b, c) \in S\}$

Chapter 2

Properties of categories

2.1 Proving simple properties

Simply defining a category is not enough. We must prove that the definitions said category hold.

2.1.1 Example 1: composition of group homomorphisms

Given: $(G, \bullet), (G', \bullet'), (G'', \bullet'')$ and $G \xrightarrow{f} G' \xrightarrow{g} G''$

Prove: $\forall a, b \in G \,.\, (g \circ f)(a \bullet b) \stackrel{?}{=} (g \circ f)a \bullet (g \circ f)b$

$$\begin{aligned}(g \circ f)(a \bullet b) &= g(f(a \bullet b)) \\ &= g(f(a) \bullet' f(b))) \\ &= g(f(a)) \bullet'' g(f(b)) \\ &= (g \circ f)a \bullet'' (g \circ f)b\end{aligned}$$

2.1.2 Example 2: associativity of matrix multiplication

Given: $m \xrightarrow{A} n \xrightarrow{B} p \xrightarrow{C} l$ and $A \xrightarrow{R} B \xrightarrow{S} C \xrightarrow{T} D$

Prove: $(R; S); T \stackrel{?}{=} R; (S; T)$

(a note on notation: $(matrix^{col}_{row} \mid {}^{row<m}_{col<n})$)

We know $(a^j_i \mid {}^{i<m}_{j<n})\ (b^k_j \mid {}^{j<n}_{k<p}) = (\sum_{j<n} a^j_i b^k_j \mid {}^{i<m}_{k<p})$

For any row i and column j, the (i,j)th entry of $(R;S);T$ is:

$$\begin{aligned}(R;S);T &= (\sum_{j<n} a_i^j b_j^k)(T) \\ &= \sum_{k<p}(\sum_{j<n} a_i^j b_j^k)(c_k^q) \\ &= \sum_{k<p}\sum_{j<n} a_i^j b_j^k c_k^q \\ &= \ldots \quad \text{(form is symmetrical at this point)}\end{aligned}$$

2.2 The category which is the product of two categories

The category the product of categories $C \times D$ consists of:

- An object (X,Y) such that $X \in C, Y \in D$
- The morphism $(X,Y) \to (W,Z) = (f,g)$ where
  $f : X \to W \in C$
  $g : Y \to Z \in D$
- The identity morphism $id_{(X,Y)} = (id_X, id_Y)$
- The composite $(X,Y) \xrightarrow{(f,g)} (W,Z) \xrightarrow{(h,k)} (U,V) = (f;k, g;k)$

2.3 The opposite (dual) category

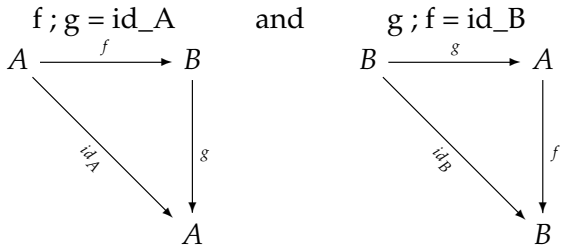
The dual category of C is C^{op}, and consists of:

- All objects $X \in C$
- Morphisms $X \to Y \in C^{op}$ are morphisms $Y \to X \in C$
- The identity morphism $id_X : X \to X \in C^{op}$ is the morphism $id_X \in C$
- Compositions $X \underset{X,Y,Z}{;} Z \in C^{op}$ are compositions $Z \underset{Z,Y,X}{;} X \in C$

2.4 Isomorphisms

2.4.1 The inverse morphism

The inverse of $f : A \to B$ is $g : B \to A$ such that:



(Theorem) if f has inverses g and g', then $g = g'$

2.4.2 Isomorphisms

An isomorphism is a morphism with an inverse. If f is an isomorphism, then $f : A \cong B$

For example:

- In **Set**, all bijections are isomorphisms
- In **Mat**, all non-singular matrices are isomorphisms
- In the category of towns, all identities id_X are isomorphisms, but no other morphism is an isomorphism. This is because identity routes are single item lists, and composing two routes (concatenating routes) can never be done in a way such that the length of the resulting route decreases. i.e. there are no inverses.

  Given we need a composition that results in a route of length 1, no morphism other than single town routes (identity routes) will do.

2.5 Initial and terminal objects

An object $X \in C$ is **initial** when for all $Y \in C$ there is a unique function $X \to Y$. i.e. $\forall Y \in C \,.\, \exists! f : X \to Y$

An object $X \in C$ is **terminal** when for all $Y \in C$ there is a unique function $Y \to X$. i.e. $\forall Y \in C \,.\, \exists! f : Y \to X$

For example, in **Set**:

- There is exactly one initial object, which is the **empty set**. This is because there is only one function from the empty set to any other set, the **empty function**, $f_A : \varnothing \to A$.
- Every singleton set is a terminal object. This is because there is a unique function $f_1 : A \to 1$ where $1 = ()$ for any set A. We know this because there is only a single item in 1 which elements in any set can map to.

Note that there can be a function with an **empty domain**, but there cannot be a function with **non-empty domain** and **empty co-domain** since there wouldn't be anything to map to from the domain. The standard set-theoretic way to define functions explains this:

1. **Cartesian product** : $A \times B = \{(a, b) \mid a \in A, b \in B\}$

   The product is a set of ordered pairs.

2. **Relation** : $R \subset A \times B$ where R is a relation between sets A and B, often written as $a\, R\, b$

3. **Function** : $f : A \to B$ is a relation such that:

   - **there exist images** : $\forall a \in A \,.\, \exists b \in B \,.\, (a, b) \in f$
     This is different to **surjectivity**, which states that everything in B is an image. This condition requires the existence of images. Surjectivity is vacously true for empty sets.
   - **images are unique** : $\forall a \in A \,.\, (a, b) \in f \wedge (a, b') \in f \Rightarrow b = b'$.

Since the only subset of $\varnothing$ is $\varnothing$, the only relation between A and B (where one of them is $\varnothing$) is $\varnothing$. Thus, the question is whether the empty relation ($\varnothing$) is a function from A to B:

- $A \neq \varnothing$: no, since there exists an $a \in A$ but no $b \in B$ such that $(a, b) \in \varnothing$, the image existence condition is not matched.
- $A = \varnothing$: yes, both existence and uniqueness conditions are vacuously true. i.e. there is no $a \in A$ to disprove the universals.

Note that the conditions state that functions must define a unique output (image) for every input (preimage). This is vacuously true on the input, since there is no input.

Also note that in many languages (such as Java, C, OCaml) there might not be function definition for the signature $\varnothing \to T$. A program without input is a function from the unit set ($1 = \{()\}$). Therefore, function $f : \varnothing \to T$ cannot be defined, but is still a function.

2.6 Exercises

2.6.1 Theorem : every inverse morphism is unique

If $g, g' : D \to C$ are two inverses for $f : C \to D$, then $g = g'$

i.e. $(g \circ f = id_C \wedge f \circ g = id_D) \wedge (g' \circ f = id_C \wedge f \circ g' = id_D) \Rightarrow g = g'$

Proof:

$$
\begin{aligned}
g &= id_C \circ g \\
&= (g' \circ f) \circ g \\
&= g' \circ (f \circ g) \\
&= g' \circ id_D \\
&= g'
\end{aligned}
$$

Alternatively:

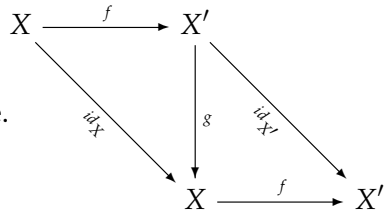
$$\begin{aligned} g &= g; id_C \\ &= g; (f; g') \\ &= (g; f); g' \\ &= id_D; g' \\ &= g' \end{aligned}$$

2.6.2 Theorem: initial objects have unique isomorphisms between them

If $X, X' \in C$ are initial, there is a unique isomorphism $X \cong X'$

Proof: Given $f : X \to X'$ and $g : X' \to X$

$X \xrightarrow{f} X'$	is a unique morphism because X is initial
$X' \xrightarrow{g} X$	is a unique morphism because X' is initial
$X \xrightarrow{g \circ f} X$	is unique because X is initial, so any morphism from X is unique
$f; g = id_X$	because identity on X must exist, and $f; g$ is unique
$g; f = id_{X'}$	because identity on X' must exist, and $g; f$ is unique
	$\therefore g$ and f are inverses
	$\therefore X \cong X'$ is a unique isomorphism since f and g are unique
	$\therefore X$ and X' are **unique up to isomorphism**.



i.e. each morphism is unique, meaning $f; g = id_X$

If objects are **unique up to isomorphism**, they are the same object with different names for things. More formally, all the objects satisfying a given definition are isomorphic.

2.6.3 Theorem: terminal objects have unique isomorphisms between them

If $X, X' \in C$ are terminal, there is a unique isomorphism $X \cong X'$

The proof would have the same structure as the equivalent theorem for initial objects. Only difference would be that the unique morphisms defined are the other way round.

2.6.4 Initial and terminal objects in the category of groups, Grp

(Initial object)

The trivial group $(1, *) = \{e\}$ (group consisting of only the identity element) is an initial object of **Grp**.

First prove $\forall G \in \mathbf{Grp} \,.\, \exists!\, 1 \to G$:

Given group $(G, \bullet)$ with identity e_G, we can define a function f

$$\begin{array}{rl} f(e) = e_G & \text{because all group homomorphisms preserve identity} \\ \forall G \in \mathbf{Grp} \,.\, \exists!\, 1 \to G & \text{because the morphism defined by } f \text{ that maps } e \text{ to } e_G \text{ is unique} \end{array}$$

Then prove $\forall G \in \mathbf{Grp} \,.\, 1 \to G$ is a valid morphism:

Show that $f(a) \bullet f(b) = f(a * b)$

$$\begin{array}{lll} f(e) \bullet f(e) & = e_G \bullet e_G & \text{by definition of } f \\ & = e_G & \text{by definition of the identity element} \\ & = f(e) & \\ & = f(e * e) & \end{array}$$

(Terminal object)

The trivial group $(1, *)$ is also a terminal object in **Grp**.

First, by same argument that makes 1 initial, we show that $\forall G \in \mathbf{Grp} \,.\, \exists!\, G \to 1$. The mapping is defined by

$$\forall g \in G \,.\, f(g) = e$$

Then, we verify that $G \to 1$ is a valid group homomorphism:

Show that $\forall a, b \in G \,.\, f(a) * f(b) = f(a \bullet b)$:

Given $a \bullet b = c$

$$\begin{array}{lll} f(a) * f(b) & = e * e & \text{by definition of } f \\ & = e & \text{by definition of the identity element and } * \\ & = f(c) & \text{by definition of } f, \text{ we know } f(c) = e \\ & = f(a \bullet b) & \text{because } a \bullet b = c \end{array}$$

Chapter 3

Functors and more categories

3.1 Morphism with no inverse in the category of posets and monotone functions

Unlike **Set**, where bijections are isomorphisms, in the category of **posets and monotone functions**, not every bijection is an isomorphism.

For instance, given the following sets:

$A = \{1, 2\}$ where A is discrete

$B = \{1, 2\}$ where $1 \leq 2$

$A \to B$ could map $2_A \mapsto 1_B$ and $1_A \mapsto 2_B$, which is valid because elements in A are disconnected.

$B \to A$ would not be able to map $1_B \mapsto 2_A$ and $2_B \mapsto 1_A$ (in reverse) and keep order because $2_A \not\leq 1_A$

3.2 Initial and terminal objects for the category of posets and monotone functions

(Initial object)

When we regard a **poset** as a category, then the initial object is the **smallest element**, if it exists. This is because there is a single morphism between the smallest object and any other object; the smallest object is always smaller than any other object.

In the **category of posets**, however, we have to consider the poset for which there is a unique monotone function between it and every other object.

Analogous to **Set**, this would be the **empty poset** $(\varnothing, \varnothing)$, which is discrete, indiscrete, and has no relations (is a flat poset).

(Terminal object)

When we regard a **poset** as a category, then the terminal object is the **largest element**, if it exists. This is because there is a single morphism from any object to the largest object; the largest object is always larger than any other object.

In the **category of posets**, analogous to **Set**, this would be the **singleton equality poset** $(1, =)$.

3.3 Functors

A functor $F : \mathcal{C} \to \mathcal{D}$ is a function that maps between categories $\mathcal{C}$ and $\mathcal{D}$ by mapping all definitions from $\mathcal{C}$ to $\mathcal{D}$ such that:

1. all objects in $\mathcal{C}$ are mapped to all objects in $\mathcal{D}$ by F

   $ob\mathcal{C} \xrightarrow{F} ob\mathcal{D}$ i.e. $X \in \mathcal{C}$ then $F(X) \in \mathcal{D}$

2. all morphisms in $\mathcal{C}$ are mapped to a morphism in $\mathcal{D}$

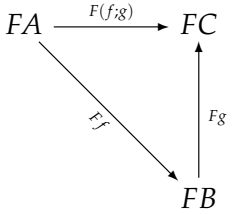
   $\forall A, B \in \mathcal{C} \wedge f : A \to B \,.\, FA \xrightarrow{F_{A,B}f} FB$

3. the identity morphisms are also mapped and maintained

   $\forall A \in \mathcal{C} \,.\, Fid_A = id_{FA}$

4. composition is also mapped and maintained

   $\forall A, B, C \in \mathcal{C} \wedge A \xrightarrow{f} B \xrightarrow{g} C \,.\, F(f \underset{A,B}{;} g) = Ff \underset{FA,FB}{;} Fg$



Functors can also be thought of as homomorphisms between categories.

3.4 More categories

3.4.1 Monoids

Any monoid $(M, e, \bullet)$ gives rise to a category $\tilde{M}$ if:

- $ob\tilde{M} \stackrel{\text{def}}{=} 1$ where 1 is any singleton set. e.g. the unit set $\{()\}$
- $\tilde{M}((), ()) \stackrel{\text{def}}{=} M$ morphisms in $() \to () \in \tilde{M}$ are the elements of M

- $id_{()} \stackrel{\text{def}}{=} e$ the identity morphism is the identity element $e \in M$
- $() \xrightarrow{m} () \xrightarrow{m'} () \stackrel{\text{def}}{=} m \bullet m'$ composition is the monoid binary relation $(\bullet)$, e.g. concatenation

3.4.2 Preord and preordered classes

Preord is the category of preordered sets and monotone maps.

Any preordered class (a class can be as big as the universe of all sets) $(A, \leq)$ gives rise to a category $\hat{A}$ if:

- objects in $\hat{A}$ are elements in A

  $ob\hat{A} \stackrel{\text{def}}{=} A$
- the set of morphisms are either the singleton set or the empty set

  $$\forall x, y \in A \,.\, \hat{A}(x, y) \stackrel{\text{def}}{=} \begin{cases} 1 & x \leq y \\ \varnothing & x \not\leq y \end{cases}$$
- the identity morphism is the element of the singleton set

  $id_x \stackrel{\text{def}}{=} ()$
- composition is the element of the singleton set, so all objects are related by this element.

  $x \xrightarrow{()} y \xrightarrow{()} z \stackrel{\text{def}}{=} ()$

3.4.3 Groupoid

A **groupoid** is a category where every morphism is an isomorphism.

The category of sets and bijective functions, for instance, is a **groupoid**.

3.5 Functor examples

3.5.1 Revisiting the inverse of town routes

To solve the problem for whether any non-identity morphism has an inverse in the category of towns and routes, we can define the following functor:

$F : \textbf{Town} \to \textbf{Nat} = \widetilde{(\mathbb{N}, 0, +)}$ i.e. from **Town** to the monoid of natural numbers

- $\forall\, town \in \textbf{Town} \,.\, town \mapsto () \in \textbf{Nat}$

- $(f : x_0 \to x_n = x_0 \xrightarrow{f=[x_0, \ldots, x_n]} x_n) \mapsto n \in \mathbf{Nat}$
- $(id_{x_0} = x_0 \xrightarrow{[x_0]} x_0) \mapsto 0 \in \mathbf{Nat}$
- $(f; g = x_0 \xrightarrow{f=[x_0, \ldots, x_n]} x_n \xrightarrow{g=[x_n, \ldots, x_m]} x_m) \mapsto n + m \in \mathbf{Nat}$

Isomorphisms are preserved by functors.

Morphisms are equal to the route lengths in **Nat**.

Composition is concatenation of routes in **Town**, which translates to addition in **Nat**.

An identity morphism is the singleton route in **Town** and zero (0) in **Nat**.

$\therefore$ Since adding (composing) any number with a non-zero natural number (morphism) in **Nat** will never produce the identity (0), and **Town** is mapped to **Nat** by F (preserving isomorphisms), concatenating two routes that are not the identity will never produce the identity route either. Thus, no non-identity route has an inverse.

3.5.2 Define a functor $C^{op} \times C \to$Set—the hom-functor

Given category C, define functor $F : C^{op} \times C \to$ **Set**

First, we know category $C^{op} \times C$ consists of:

- **(objects)** $(X, Y) \in C^{op} \times C$ where $X \in C^{op}$ and $Y \in C$.
  i.e. $X, Y \in C$ since objects in C are in C^{op}
- **(morphisms)** $(X, Y) \mapsto (X', Y') \in C^{op} \times C$ is (f, g) where
  $f : X \to X' \in C^{op}$
  $g : Y \to Y' \in C$
- **(identities)** $id_{(X,Y)} = (id_X, id_Y)$
- **(composition)** $(f, g); (f', g') = (f; f', g; g')$ where
  $f; f' \in C^{op}$ or $f'; f \in C$
  $g; g' \in C$

Now we define what we map to in **Set**:

- **(objects)** $(X, Y) \in (C \times C^{op}) \mapsto hom_C(X, Y) \in$ **Set**
- **(morphisms)** for **Set**, a morphism is all functions $hom_C(X, Y) \to hom_C(X', Y')$
  Given (f, g) where
  $f : X \to X' \in C^{op}$ i.e. $f : X' \to X \in C$
  $g : Y \to Y' \in C$

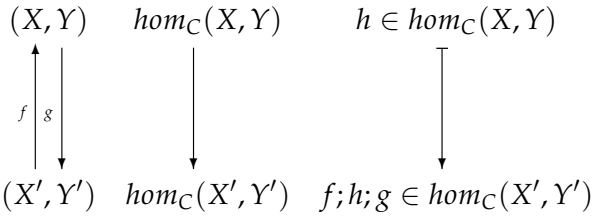
We can define a morphism (functions) $hom_C(X,Y) \to hom_C(X',Y') \in$ **Set** as follows:

if $h \in hom_C(X,Y)$

then $f;h;g \in hom_C(X',Y')$

so $h \mapsto f;h;g$

In short:



Now we check it is a valid functor:

- **(identities)** $(id_X, id_Y) \mapsto h \mapsto id_X;h;id_Y = id_{hom_C(X,Y)}$
- **(composition)** Given $h \in hom_C(X,Y)$

  We want to show that the following composition is valid:

  $hom_C(X,Y) \xrightarrow{A} hom_C(X',Y') \xrightarrow{B} hom_C(X'',Y'')$

  $h' = f;h;g$ and $f';h';g' = f';f;h;g;g'$

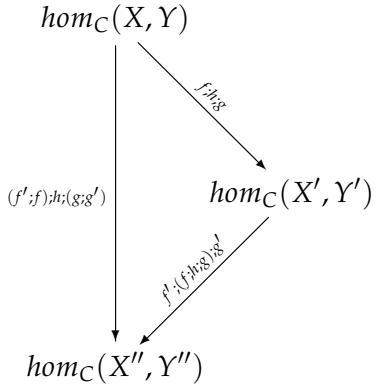
  we know that composition $A;B : hom_C(X,Y) \to hom_C(X'',Y'') = f';f;h;g;g'$

  We now prove composition is maintained by the functor:

  Given $(f,g) = (X' \to X'', Y \to Y')$ and $(f',g') = (X \to X', Y' \to Y'')$

  $F((f;g)\ ;\ (f';g'))$
  $= F(f';f\ ,\ g;g')$
  $= (f';f;h;g;g')$ where $h \in hom_C(X,Y)$
  $= f';(f;h;g);g'$
  $= f';F(f,g);g'$
  $= F(f,g);F(f';g)$

  i.e.



3.6 Homework exercises

3.6.1 Define a functor from C to a preorder

Every category can be turned into a preordered class.

Given category C, write a preorder $X \leq Y$ when $\exists X \to Y$

(Part 1) Prove $\leq$ is a preorder on obC:

A preorder is a reflexive and transitive binary relation $\leq$.

- reflexive because for any $X \in C$, id_X is a morphism $X \to X$, so $X \leq X$ exists and is reflexive.
- transitive because if $X \leq Y \leq Z$ then we can pick a map $f : X \to Y$ and a map $g : Y \to Z$ so $f;g$ is a map $X \to Z$. This morphism is mapped to $X \leq Y$, which means the composition is transitive.

For instance, in **Set**, $A \leq B$ if B is non-empty or A is empty.

(functions are left-total and single-valued)

(Part 2) Define a functor $C \to \widehat{C} = (\widehat{obC, \leq})$

$F : C \to \widehat{obC, \leq}$

- $ob\widehat{C} \stackrel{\text{def}}{=} obC$ i.e. $\forall X \in C \,.\, X \in C \mapsto X \in \widehat{C}$
- Given $X \xrightarrow{f} Y \in C$, we know $\widehat{C}(X, Y) \stackrel{\text{def}}{=} 1$ i.e. $\forall X \leq Y \in C \,.\, f \mapsto ()$

therefore:

$$\begin{array}{ccc} A & & FA = A \\ \downarrow & & \downarrow \\ F & \longmapsto & () \\ \downarrow & & \downarrow \\ B & & FB = B \end{array}$$

We know this is valid because:

- $id_X \stackrel{\text{def}}{=} ()$ i.e. $\forall x, id_X \in C \,.\, id_X \mapsto ()$
- $X \xrightarrow{()} Y \xrightarrow{()} Z \stackrel{\text{def}}{=} ()$ i.e. composition is maintained trivially,

Additionally, since this definition describes a category that rises from a preordered class (section 3.4.2), we can argue from the resulting category that C is a preordered class.

3.6.2 Full and faithful

Given $F_{X,Y} : hom_C(X, Y) \to hom_D(F(X), F(Y))$

- A **full** mapping is one which is **surjective**. (left-total)

  So $F_{X,Y}$ is full if it is surjective.

  i.e. every morphism in $F\ C$ is mapped to by at least one morphism of C.

  symbolically: $\forall g \in F\ C\ .\ \exists f \in C\ .\ g = F(f)$

- A **faithful** mapping is one which is **injective**.

  *note: injective and single-valued are not the same.

  i.e. $F_{X,Y}$ is faithful when for every $X, Y \in C$, $f : X \to Y$ and $g : X \to Y$, $F\ f = F\ g$ implies $f = g$

  in short: $\forall f, g \in C\ .\ F(f) = F(g) \Rightarrow f = g$

- $F_{X,Y}$ is **fully faithful** if it is both full and faithful.

Note that these definitions only care about morphisms being mapped.

Is the functor defined in 3.6.2 faithful and/or full?

We check:

- The functor is full because:

  $\forall g : FX \to FY \in D$

  $\exists f : X \to Y \in C$

  such that $F_{X,Y}(f) = g$

  since g impies $X \leq Y$ and $X \leq Y$ implies there is an $f : X \to Y$, then $f \mapsto g$, so every f is mapped.

- The functor is not faithful for C if $C(X, Y) = \{f, g, h\}$. This is because all morphisms will be mapped to the same thing, the empty tuple.

  So, $f, g, h \mapsto ()$, meaning, $F_{X,Y}$ is not single valued. However, is this the case for every C?

  To prove this is not the case, we can define a category D where this is an injective mapping. e.g. Given $hom_D(X, Y) = k, k \mapsto ()$. This is injective.

  Therefore, F is **not faithful in general**.

3.6.3 Prove category $(C(X, X), id_X, ;_{X,X})$ is a monoid

Given category C with object X, the monoid $M = (C(X, X), id_X, \underset{X,X}{;})$ is defined as follows:

- id_X is the identity element for morphisms

- $;_{X,X}$ (composition) is an associative binary operator by definition (it comes from a category, where–by definition–composition is associative)

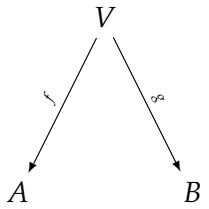
Chapter 4

Cones, Products and Sums

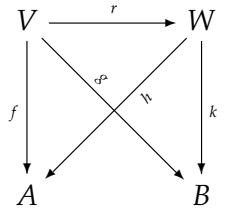
4.1 The category of cones

Given $A, B, V, W \in \mathcal{C}$, cones over $A, B \in \mathcal{C}$ form a category $Cone(\mathcal{C}, A, B) or Cone_{\mathcal{C}}(A, B)$.

A **cone** from V to A and B consists of:



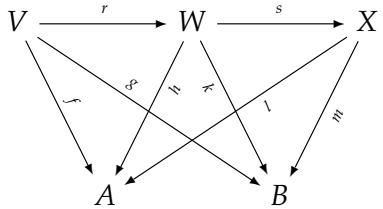
A **cone morphism** from (V, f, g) to (W, h, k) is a $\mathcal{C}$ morphism $V \xrightarrow{r} W$ such that the two triangles, from V through W to A and to B, commute.



f and $r; h$ are equal, as do g and $r; k$, i.e. $r; h = f$ and $r; k = g$.

If a diagram commutes, any two paths from one object to another are equal.

Given $r : V \to W$ and $s : W \to X$, composition is the same as composition in $\mathcal{C}$, with the added check for commutativity of the triangles.



i.e. $r;s;l = f$ and $r;s;m = g$.

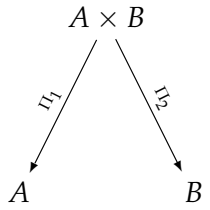
$f = r;h \wedge h = s;l$
$\therefore f = r;s;l$

$g = r;k \wedge k = s;m$
$\therefore g = r;s;l$

4.2 Product object (categorical product) in a category of cones

A product of A and B is a "terminal cone" over A and B; i.e. a terminal object in $Cone_{\mathcal{C}}(A, B)$.

Given **projections** $\Pi_1 = (a, b) \mapsto a$ and $\Pi_2 = (a, b) \mapsto b$

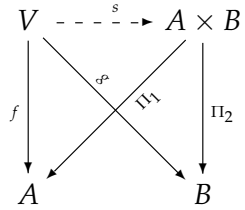


Where Π_1 and Π_2 are called the left and right projections respectively.

To show that there is a unique morphism to $A \times B$, we must prove that such morphism exists, and is unique.

For example, in **Set**:

We start by defining a unique a morphism from V to $A \times B$.



Given $x \in V$, we can define $h = x \mapsto (fx, gx)$, proving such morphism indeed exists.

We also know $\Pi_1(s(x)) = f(x)$ and $\Pi_2(s(x)) = g(x)$ because s is a cone morphism and paths much commute.

Hence, we know s a unique morphism. i.e. $s;\Pi_1 = f = h;\Pi_1$ and $s;\Pi_2 = g = h;\Pi_2$, $\therefore$ $s = h$

4.3 The sum of sets

The sum of sets A and B is defined by a union:

$A + B = \{\textbf{inl}\ x \mid x \in A\} \cup \{\textbf{inr}\ y \mid y \in B\}$

where **inl** and **inr** are injective: $\forall x, y.\textbf{inl}\ x \neq \textbf{inr}\ y$

Thus $\textbf{inl}\ x = (0, x)$ and $\textbf{inr}\ y = (1, y)$

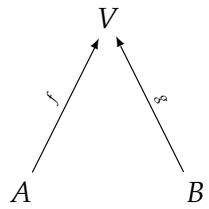
note that this holds, even for $A + A$ or $\mathbb{N} + \mathbb{N}$.

e.g. $\textbf{inl}\ 3 \neq \textbf{inr}\ 3 \in \mathbb{N} + \mathbb{N}$

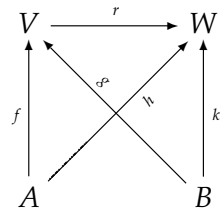
Note: the set of functions $A \to B$ can be written in notation B^A. This is because the number of elements in B^A is m^n where $m = size(B), n = size(A)$.

4.4 The category of cocones

Let $A, B \in C$, a **cocone** over A, B is the dual notion of a **cone**, and defined by the following diagram:



So a morphism from cocone (V, f, g) to cocone (W, h, k) is R and defined by the following diagram:



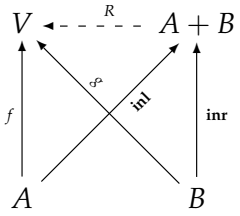
Composition is defined by the dual concept of composition in cones.

We therefore obtain a category $Cocone_C(A, B)$ of cones over A, B.

4.4.1 Coproducts and initial cones in Set

The **coproduct** (sum defined in section 4.3) $A + B$ is an "initial cone" in $Cocone_{Set}(A, B)$.

To show $A + B$ is initial in $Cocone_{Set}(A, B)$, we must show there is a unique morphism $R : A + B \to V$.



To do this, we define a function s given $x, y \in A + B$:

$$s = \begin{cases} \textbf{inl}\ x \mapsto fx \\ \textbf{inr}\ y \mapsto gy \end{cases}$$

We know s is equal to R, and hence a unique morphism, because:

- **inl**; s and f commute
- **inr**; s and g commute

Given the diagram commutes, because it's a cone morphism, the morphisms are all equal there is a unique function.

4.4.2 The dual notion

A coproduct in C for A, B is a product in C^{op} for A, B.

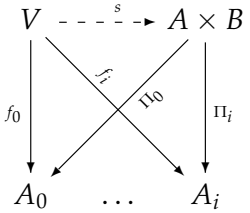
4.5 Generalising from binary to arbitrary

4.5.1 Generalised categorical products

Generalised products are denoted by the **categorical product** of the I indexed family A_i where $i \in I$, and I is a set, countable or uncountable, that indexes the family. These are n-ary products if I is of size n.

This is given by $\prod_{i \in I}(A_i)$, where in composition, commutativity of all triangles formed commute.

$\forall i \in I\ .\ f_i = r; \Pi_i$ in the following diagram:



In **Set**: A way to define general product is:

$\prod_{i \in I}(A_i) \stackrel{\text{def}}{=}$ the set of functions p with domain I such that $\forall i \in I\ .\ p_i \in A_i$

Here, the i^{th} projection would be $p \mapsto p\ i$, i.e. the i^{th} element of p, or applying p to i.

4.5.2 Generalised sum of sets

The **categorical coproduct** of $(A_i)_{i \in I}$ is given by $\sum_{i \in I} A_i = \{(i, a) | i \in I \wedge a \in A_i\}$

Hence, the categorical product is a big tuple/function, and the categorical coproduct is a big disjoint-union/sum.

Another notation used for coproducts is $\coprod_{i \in I}(A_i)$

4.5.3 Example: product and sum of the empty family

In **Set**, the Π of the empty family $(\varnothing \to A)$ is the singleton set, so the set of cardinality of the product is 1.

$\prod_{\varnothing} = \{f_{\varnothing} : \varnothing \to \varnothing\} = \{()\} = 1$

The sum of the empty family is the empty set.

$\sum \varnothing = \varnothing$

4.5.4 Common patterns with monoids

The patterns observed with the product and sum of the empty set are analogous with different monoids.

For instance, for lists:

- $Sum\ [] = 0 \quad (\mathbb{N}, 0, +)$
- $Product\ [] = 1 \quad (\mathbb{N}, 1, \times)$
- $Max\ [] = 0 \quad (\mathbb{N}, 0, max)$
- $Min\ []$ undefined (infinity if analogous to division by zero, assymptotes, etc.)
- $And\ [] = True \quad (\mathbb{B}, True, \wedge)$
- $Or\ [] = False \quad (\mathbb{B}, False, \vee)$

For these monoids, it is a common pattern that vacuous arguments return the identity as a convention. Note that *Min* is an example of a non-monoid, and thus, can't return the identity, making the function undefined on vacuous arguments.

Thus, if there is a monoid $(M, e, *)$, and we want an n-ary application of $*$ on the elements M, the application must end with e.

Hence, every n-ary application following a monoidal structure returns the identity by convention.

Chapter 5

Functors and Naturality

Category theory was initially introduced in the 1940s as a way to define naturality. Before looking into that, however, we define more examples of functors.

5.1 More functors

5.1.1 Example 1: an arbitrary functor

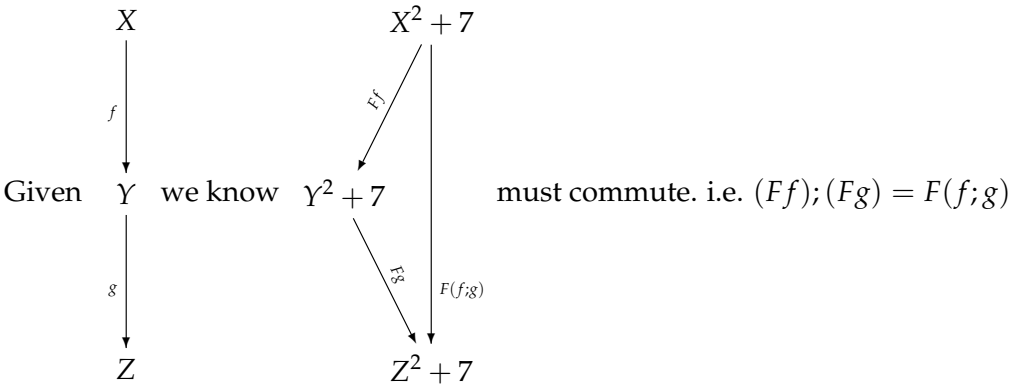
We can define functor F : **Set** $\to$ **Set** by the following mappings:

- objects: $X \mapsto X^2 + 7$, where $7 = \{0, 1, 2, 3, 4, 5, 6\}$ is the set of seven objects.
- morphisms: $g = Ff$ in $\begin{array}{ccc} X & \xrightarrow{F} & X^2+7 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ Y & & Y^2+7 \end{array}$ e.g. $\begin{array}{ccc} Towns & \xrightarrow{F} & Towns^2+7 \\ {\scriptstyle population}\downarrow & & \downarrow{\scriptstyle F\ population} \\ \mathbb{N} & & \mathbb{N}^2+7 \end{array}$

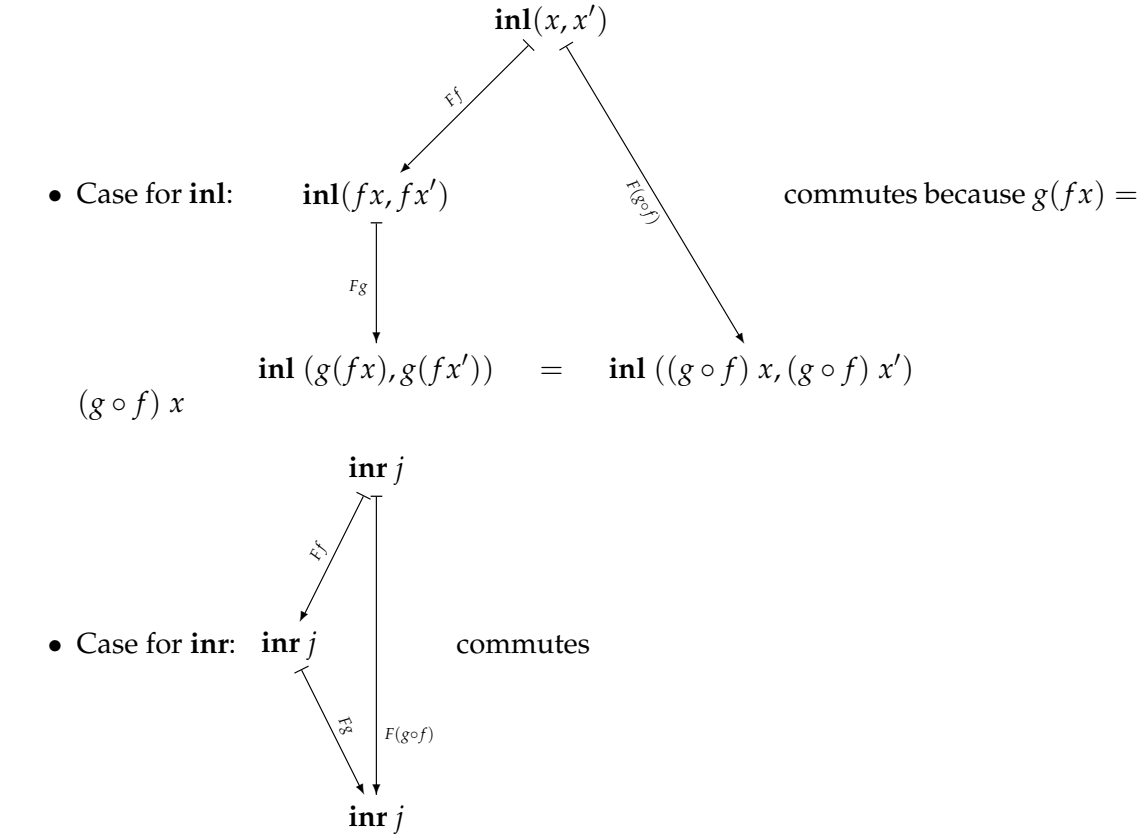
Given the morphism $f \in$ **Set** we can define the mapping as follows:

$$F\,f = \begin{array}{c} X^2+7 \\ \downarrow{\scriptstyle F\,f} \\ Y^2+7 \end{array} = \begin{array}{c} \mathbf{inl}(x, x') \\ \downarrow{\scriptstyle Ff} \\ \mathbf{inl}(f\ x, f\ x') \end{array} + \begin{array}{c} \mathbf{inr}\ j \\ \downarrow{\scriptstyle Ff} \\ \mathbf{inl}\ j \end{array}$$

Hence, composition is as follows:

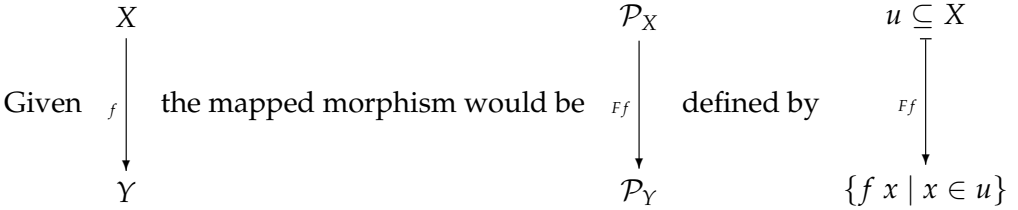


To check composition works, we check the each sum case (**inl** and **inr**):



5.1.2 Example 2: Powerset

$F : X \mapsto \mathcal{P}_X$ (powerset of X)



Objects in the powerset are subsets of X. Thus, u is any subset of X and is mapped to any subset of Y.

5.1.3 Example 3: List

F : **Set** $\to$ **Set**

- Objects: $X \overset{F}{\longmapsto}$ **List** X
- Morphisms: $\begin{array}{ccc} X & \overset{F}{\longmapsto} & \textbf{List}\ X \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle Ff} \\ Y & & \textbf{List}\ Y \end{array}$ where Ff is defined by $\begin{array}{c} l \in \textbf{List}\ X \\ \downarrow \\ (\texttt{map} f\ l) \in \textbf{List}\ Y \end{array}$

5.1.4 Example 4: Opposite to Powerset

F : $\textbf{Set}^{op} \to$ **Set**

- Objects: $X \overset{F}{\longmapsto} \mathcal{P}_X$
- Morphisms: $\begin{array}{ccc} X & \overset{F}{\longmapsto} & \mathcal{P}_X \\ {\scriptstyle f}\downarrow & & \uparrow{\scriptstyle Ff} \\ Y & & \mathcal{P}_Y \end{array}$ where Ff is defined by $\begin{array}{c} \{x \in X \mid f\ x \in u\} \\ \uparrow \\ u \in \mathcal{P}_Y \end{array}$

Note that F is a **contravariant** functor. This definition will be coming up soon.

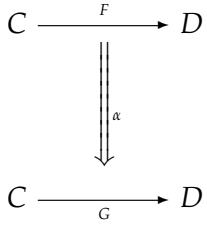
5.1.5 Example 5 : Functor $C^2 \to C$

F : $\textbf{Set}^2 \to$ **Set**

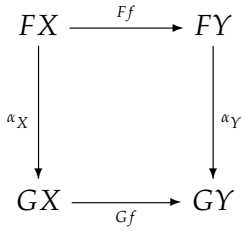
- Objects: $(X, X') \overset{F}{\longmapsto} X + X'$
- Morphisms: $\begin{array}{ccc} (X, X') & \overset{F}{\longmapsto} & X + X' \\ {\scriptstyle f}\downarrow\ {\scriptstyle f'}\downarrow & & \downarrow{\scriptstyle F(f,f')} \\ (Y, Y') & & Y + Y' \end{array}$ where $F(f, f')$ is defined by $\begin{array}{ccc} \textbf{inl}\ x \in X & & \textbf{inr}\ x' \in X' \\ \downarrow & + & \downarrow \\ \textbf{inl}\ fx & & \textbf{inr}\ fx' \end{array}$

In general, if C is a category with binary coproducts, we can define a functor $+ : C^2 \to C$.

5.2 Natural transformations

In the diagram  α is a natural transformation from F to G.

$\forall x \in C \,.\, \alpha : F \Rightarrow G$ provides a D morphism $FX \xrightarrow{\alpha_X} GX$ such that for any C morphism $X \xrightarrow{f} Y$ the following square commutes:

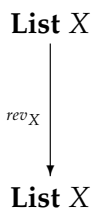
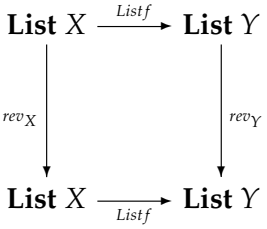


5.2.1 Example

Given the **List** functor defined before (the one that maps lists), we can show that:

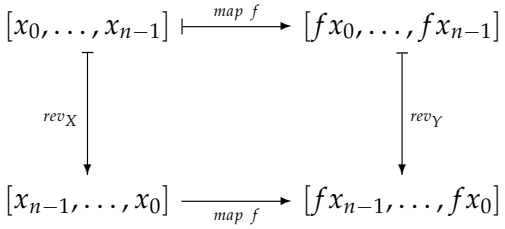
$reverse_X$: **List** $X \to$ **List** X is a natural transformation.

To do this, we must show commutativity.

Given  we must show that  commutes.

*note: $List\ f = map\ f$, which is how the *List* functor would be implemented in some programming languages (like Haskell).

The following diagram depicts this precise property:

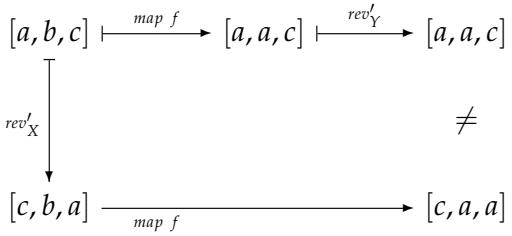


5.2.2 Non-example

The transformation (rev') which reverses a list given no duplicates is not natural. We can prove this by defining a case where the diagram does not commute. This consists of a function and a list onto which we map said function.

Given set $S = a, b, c$, we define function $f : S \to S$ to be:

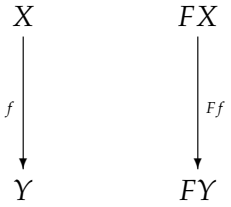
$f(a) = a$
$f(b) = a$
$f(c) = c$



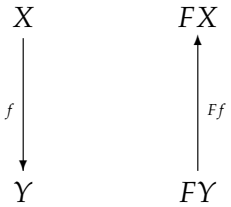
Does not commute because $[a, a, c] \neq [c, a, a]$.

5.3 Covariance and contravariance

A **covariant** functor maps to morphisms going in the same direction of the original mapped morphism.



A **contravariant** functor maps to morphisms going in the opposite direction of the original mapped morphism.



In other words, covariance preserves the ordering of types while contravariance reverses this order.

Sometimes, we refer to structures with only covariant (positive) or contravariant (negative) occurences as **functorial**, which means there is an obvious way to extend the structure to make a functor.

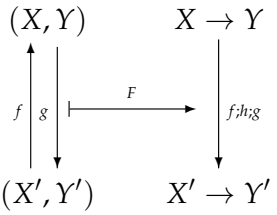
Informally, **non-functorial** structures have both positive and negative occurences. This

means there is no obvious way of extending it to become a functor. However, we don't know otherwise, i.e. there may be a way.

5.3.1 Example: arrow (function types)

The left side of an arrow is contravariant. This can be explained in category theory as the following functor:

$F : \mathbf{Set}^{op} \times \mathbf{Set} \to \mathbf{Set}$

- Objects: $(X, Y) \overset{F}{\longmapsto} X \to Y$
- Morphisms: 

where $h : X \to Y$

Note that having $\mathbf{Set}^{op}$ is the only way the function type can be defined since there would be no way to create the functor otherwise. This is analogous to $C^{op} \times C \to \mathbf{Set}$ seen before.

The implications of this is that in functions with subtypes (generally speaking, Object Oriented languages), we need to deal with left side contravariance of arrows.

e.g. Given `String` $\leq$ `Object`

- `String -> String` can be safely used wherever you expect `String -> Object`
- `Object -> String` can be safely used wherever you expect `String -> String`

As `fold` is a higher-order function, and takes one as an argument, it will have both possitive and negative occurences. Hence, it has no simple functor.

5.3.2 Example: more on programming languages

Programming languages that support subtyping must deal with variance. Variance refers to how subtyping between complex types relates to subtyping between their components.

For example, in C#:

- `Enumerable<Cat>` is a subtype of `Enumerable<Animal>`, i.e. `Enumerable<T>` is **covariant** on `T`.

- `Action<Animal>` is a substype of `Action<Cat>`, i.e. `Action<T>` is **contravariant** on `T`

This problem can be observed in Java and C# arrays. To avoid errors when creating an `Object` array, should we treat this as:

- **covariant**: `String[]` is a `Object[]`

  However, you should always be able to put `Integer` in `Object[]`, so treating `String[]` as a `Object[]` results in errors when writing.

- **contravariant**: `String[]` is a `Object[]`

  However, you should expect to only read `String` from `String[]`, so treating `Object[]` as a `String[]` results in errors when reading since `Object[]` could contain `Integer`.

- **invariant**: `String[]` and `Object[]` are different

  However, making arrays invariant rules out useful polymorphic programs.

To make arrays polymorphic, the designers of Java and C# made all arrays covariant. This, however causes runtime errors when writing into arrays, which the designers deal with using exceptions. e.g. `ArrayStoreException` in Java. This is known as "ad-hoc polymorphism". i.e. Polymorphism where we get different behaviour depending on the type.

This was until the addition of generics, which provided a method of creating polymorphic functions without relying on covariance, which allows type checking for errors at compile time rather than runtime. This is known as "parametric polymorphism". i.e. Polymorphism where we get the same behaviour regardless the type–as in functional programs.

5.4 Exercises

5.4.1 A transformation that doubles entries

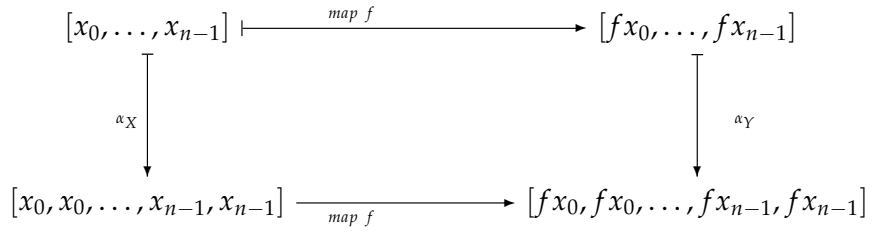
Given a transformation that doubles entries on a list:

$\alpha_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = [x_0, \ldots, x_{n-1}] \mapsto [x_0, x_0, \ldots, x_{n-1}, x_{n-1}]$

Prove it is natural.

Given $f : X \to Y$

We can show α is natural through the following square commutes:



As this is natural, behaviour does not depend on the type of the argument. Thus, this is parametrically polymorphic on **X**.

5.4.2 A transformation that only reverses lists of natural numbers

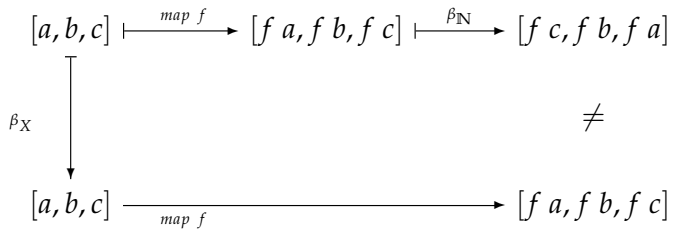
Given a transformation that reverses the list only if $\mathbf{X} = \mathbb{N}$:

$$\beta_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = \begin{cases} [x_0, \ldots, x_{n-1}] \mapsto [x_0, \ldots, x_{n-1}] & \mathbf{X} \neq \mathbb{N} \\ [x_0, \ldots, x_{n-1}] \mapsto [x_{n-1}, \ldots, x_0] & \mathbf{X} = \mathbb{N} \end{cases}$$

Prove it not is natural.

Given $f : X \to \mathbb{N}$ where $X = a, b, c$

We can show α is natural through the following square commutes:



The diagram above does not commute, thus, is not natural.

Since this is not natural, and behaviour depends on the type of the argument, in this case whether they are natural numbers, this is ad-hoc polymorphism.

5.4.3 A transformation that only reverses lists of countably infinite types

Given a transformation that reverses the list only if **X** is countably infinite:

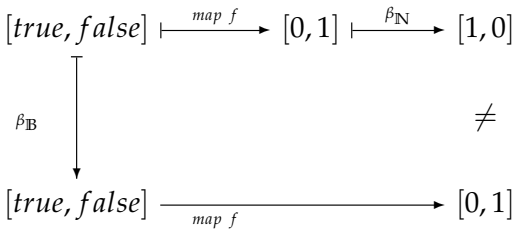
$$\gamma_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = \begin{cases} [x_0, \ldots, x_{n-1}] \mapsto [x_0, \ldots, x_{n-1}] & \mathbf{X} \text{ is not countably infinite} \\ [x_0, \ldots, x_{n-1}] \mapsto [x_{n-1}, \ldots, x_0] & \mathbf{X} \text{ is countably infinite} \end{cases}$$

Prove it not is natural.

Given $f : \mathbb{B} \to \mathbb{N}$:

$$f(b) = \begin{cases} 0 & b = true \\ \\ 1 & b = false \end{cases}$$

We can show α is natural through the following square commutes:



The diagram above does not commute, thus, is not natural. In fact, this example would also work for the previous exercise.

Chapter 6

Natural transformations and whiskered composition

6.1 More on natural transformations

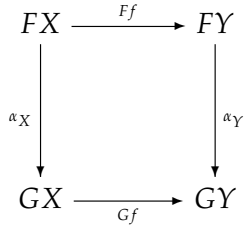
6.1.1 Natural transformations

Functors:

- map objects to objects
- map morphisms to morphisms

Natural transformations map objects to morphisms.

As defined before, a natural transformation α maps each $A \in C$ to a D morphism $\alpha_A : FA \to GA$ such that for every C morphism $f : A \to B$ the following square commutes:

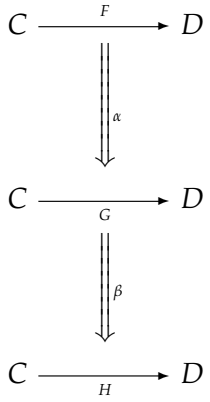


Hence many theorems use naturality–natural transformations buy us commutative diagrams which ensure everything we build commutes.

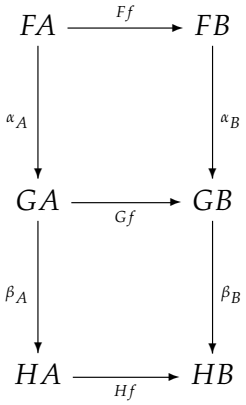
6.1.2 Composition natural transformations

The composite of α and β maps $A \in C$ to the composite D morphism $FA \xrightarrow{\alpha_A} GA \xrightarrow{\beta_A} HA$ such that it commutes.

i.e.



Is the composite of natural transformations also natural? We can check:



From the diagram above, we know the composition also commutes:

$\alpha_A; Gf = Ff; \alpha_B$
$\beta_A; Hf = Gf; \beta_B$

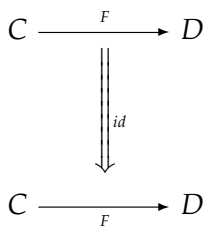
$\therefore \alpha_A; (\beta_A; Hf) = (\alpha_A; Gf); \beta_B = Ff; \alpha_B; \beta_B$

From this, we can check that it is associative.

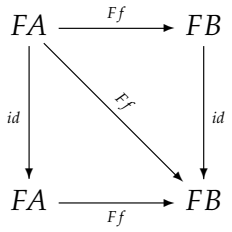
6.1.3 The identity natural transformation

The identity on F maps $A \in C$ to $FA \xrightarrow{id} FA$.

i.e.



Again, we can check this is natural:



6.1.4 A category of functors and natural transformations

It appears we can form a functor category from natural transformations $[\mathcal{C}, \mathcal{D}]$, provided $\mathcal{C}$ is small. i.e. objects are functors mapping a category $\mathcal{C}$ to $\mathcal{D}$, and morphisms are natural transformations between the functors.

- if both $\mathcal{C}$ and $\mathcal{D}$ are small, they form a small category $[\mathcal{C}, \mathcal{D}]$;
- if $\mathcal{D}$ is not small, they form a class.

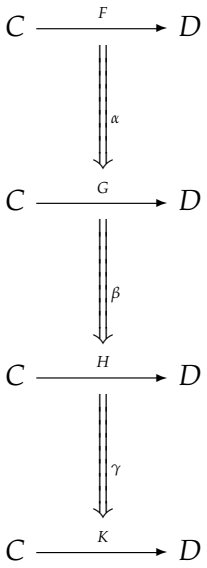
Now we check that this is a valid category:

First, we know there exists an identity, which would be the identity natural transformation seen previously. By definition, the identity composes on the right and left to produce the same morphism it was composed with.

Now, we show **composition is associative**. We know **vertical composition** is associative, even if $\mathcal{C}$ is not small. This is because morphisms are transformations, which, by definition map objects to morphisms.

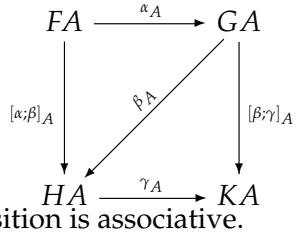
Thus, vertical composition depends on composition in $\mathcal{D}$–as $\mathcal{C}$ only provides objects– meaning it holds by definition for any transformation, even if they were not natural.

i.e.



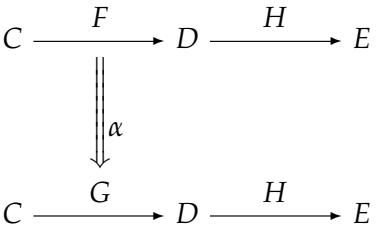
such that $(\alpha; \beta); \gamma = \alpha; (\beta; \gamma)$

Given $A \in \mathcal{C}$, associativity can be expressed with the following commutative diagram:



Which form two commutating triangles, showing composition is associative.

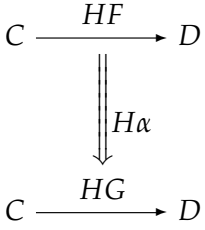
6.2 Right whiskered composition



Given a natural transformation, we can compose it with a functor to form a **right whiskered** natural transformation.

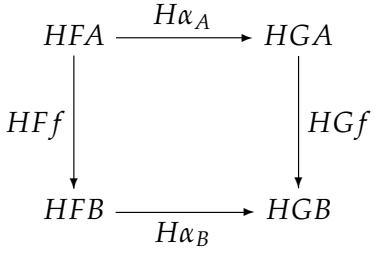
Note that this is still a natural transformation. Thus, we want to define a transformation that maps $A \in C$ to an E morphism.

i.e. $HFA \xrightarrow{H\alpha} HGA$ since

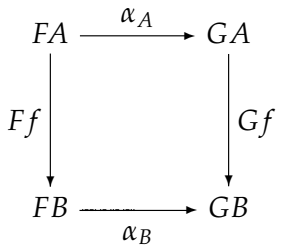


Since this is supposed to be natural, we now check it is:

Given $f : A \to B$ the following square must commute:



Which we know commutes because α is natural, meaning it buys us the following commuting square:



6.2.1 Properties of right whiskering

Apart from naturality, we must also check that the following properties hold for right whiskered compositions:

1. $C \xrightarrow{F} D \xrightarrow{H} E \xrightarrow{H'} E'$, α, $C \xrightarrow{G} D \xrightarrow{H} E \xrightarrow{H'} E'$

   $H'(H\alpha) = (H; H')\alpha$

2. $C \xrightarrow{F} D \xrightarrow{I} D$, α, $C \xrightarrow{G} D \xrightarrow{I} D$

   $I_D\alpha = \alpha$

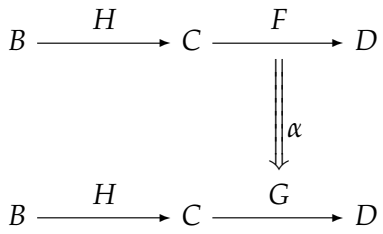
3. $C \xrightarrow{F} D \xrightarrow{K} E$, α, $C \xrightarrow{G} D \xrightarrow{K} E$, β, $C \xrightarrow{H} D \xrightarrow{K} E$

   $K(\alpha; \beta) = (K\alpha)\beta$

4. $C \xrightarrow{F} D \xrightarrow{G} E$

   $G\, id_F = id_{GF}$

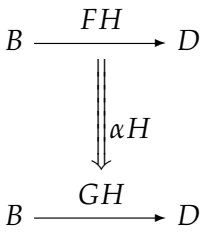
6.3 Left whiskered composition



We define the **left whiskered** natural transformation αH which maps each $X \in B$ to the D morphism

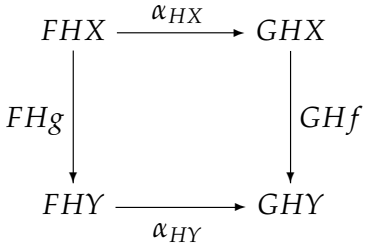
$$FHX \xrightarrow{\alpha_{HX}} GHX$$

i.e.



And again, we check this really is natural:

Given morphism $g \in B$ with type $g : X \to Y$ the following square must commute:

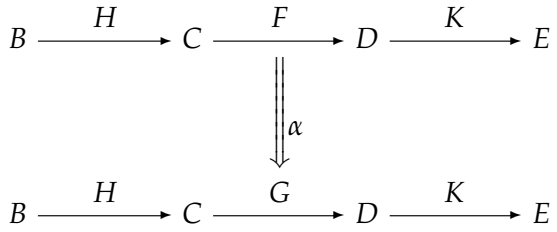


which we know commutes because α is natural at Hg.

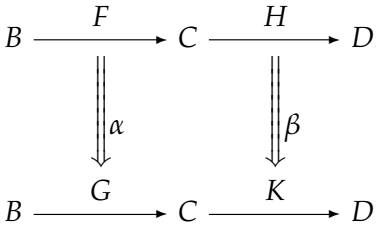
Note that although α being a natural transformation means αH is too, α does not need to be natural for αH to be.

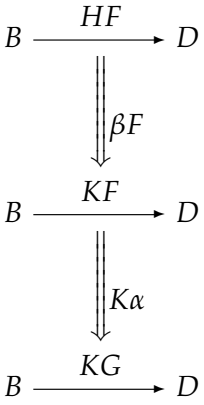
We also have to check the equivalent right whiskering properties $1, 2, 3, 4$ mentioned for left whiskering.

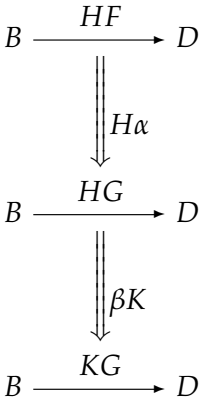
6.4 More properties of whiskered compositions

- 

  i.e. left and right whiskered compositions $\quad (K\alpha)H = K(\alpha H)$

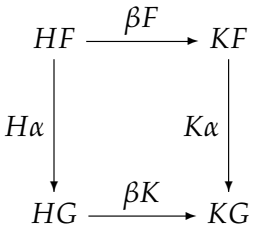
- 

i.e. the horizontal composites 

and 

are equal.

We can also say:



commutes in $[B, D]$

6.4.1 Exercise

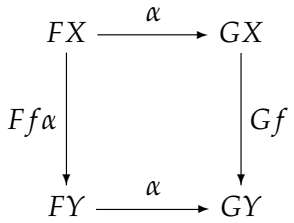
As mentioned before, α being natural ensures αH is natural. However, given an unnatural α, we can still have a natural αH.

To prove this, we first define an unnatural transformation α to show that it exists.

i.e.

$\exists X, Y \in C, f : X \to Y$

where the following diagram



does not commute; $Gf\alpha \neq Ff\alpha$.

First, we define category D with $obD = \{a, b, c, d\}$ where
$FX = a$

$FY = b$
$GX = c$
$GY = d$

For this category, we need at least 10 morphisms to have an unnatural transformation. So we have:

- $a \xrightarrow{f} b$
- $a \xrightarrow{g} c$
- $b \xrightarrow{h} d$
- $e \xrightarrow{i} d$
- $a \xrightarrow{id_a} a$
- $b \xrightarrow{id_b} b$
- $c \xrightarrow{id_c} c$
- $d \xrightarrow{id_d} d$
- $q = f; h : a \to b \to d$
- $p = g; i : a \to c \to d$

where q and p are not equal.

This is a valid category because all objects have identity morphisms, and composition is well defined.

Secondly, we define category C, with 2 objects $obC = \{X, Y\}$ and 3 morphisms:

- $X \xrightarrow{s} Y$
- $X \xrightarrow{id_X} X$
- $Y \xrightarrow{id_Y} Y$

Now we can define functors F and G:

F:

- $X \mapsto a$
- $Y \mapsto c$
- $s \mapsto g$
- $id_X \mapsto id_a$
- $id_Y \mapsto id_c$

G:

- $X \mapsto b$
- $Y \mapsto d$
- $s \mapsto h$
- $id_X \mapsto id_b$
- $id_Y \mapsto id_d$

Finally, we can define an unnatural α:

- $X \mapsto f$
- $Y \mapsto i$

Now that α is defined–and we know such a transformation does indeed exist–we want to show αH can be natural. So we need a category B such that:

$\forall x, y \in B, x \xrightarrow{w} y.$

$$\begin{array}{ccc} FHx & \xrightarrow{\alpha Hx} & GHx \\ \downarrow{\scriptstyle FHw\alpha} & & \downarrow{\scriptstyle GHw} \\ FHy & \xrightarrow{\alpha Hy} & GHy \end{array}$$

This is trivially true because universal quantifiers hold for vacuously true cases, i.e. if B is empty. Although there could be–and certainly are–more examples, just one case is enough to show that αH can be natural, even if α is not.

Chapter 7

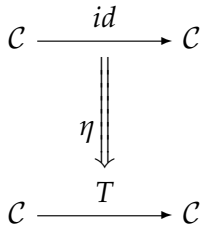
Monads

7.1 Definition 1–Using natural transformations

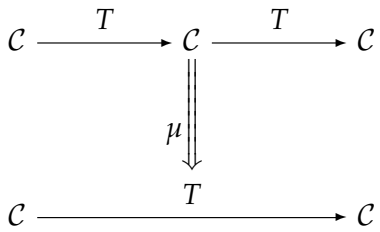
Let $\mathcal{C}$ be a category.

A monad on $\mathcal{C}$ consists of:

- a functor $T : \mathcal{C} \to \mathcal{C}$ (endofunctor)
- a natural transformation η called the unit:

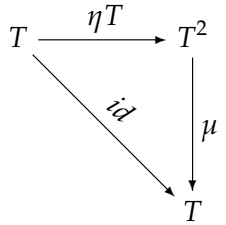


- a natural transformation μ called multiplication:

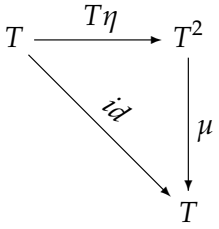


satisfying three properties:

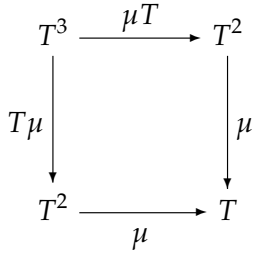
- the right identity law



- the left identity law



- the associativity law



It is similar to a monoid, so the name is not a coincidence:

- set X
- element $e \in X$
- function $* : X^2 \to X$

Also satisfying three equations:

- $\forall x \in X. x * e = x$
- $\forall x \in X. e * x = x$
- $\forall x, y, z \in X. (x * y) * z = x * (y * z)$

7.2 Definition 2–As a Kleisli triple

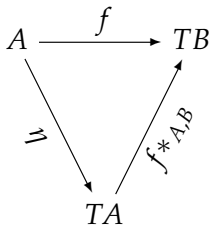
Let $\mathcal{C}$ be a category.

A Kleisli triple on $\mathcal{C}$ consists of:

- for each object A, we have an object TA and a morphism $\eta_A : A \to TA$.
- for objects A, B and morphism $f : A \to TB$, we have a morphism $f* : TA \to TB$.

such that the following equations are satisfied:

- left identity law:



analogous to:

$a : A, f : A \to TB \vdash \texttt{return}\ a >>= f = f\ a$

in Haskell.

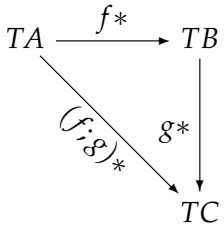
- right identity law: $\eta_{B} *_{B,B} = id_{TB}$

  analogous to:

  $p : TB \vdash p >>= \lambda x.\ \texttt{return}\ x = p : TB$

  in Haskell.

- associativity law:



  analogous to:

  $p : TA, f : A \to TB, g : B \to TC$
  $\vdash (p >>= f) >>= g = p >>= (f >>= g)$

  in Haskell.

Even though the Kleisli triple makes no mention of naturality, it is equivalent to the previous definition given.

7.3 Example: Maybe and Exception monad

$Maybe\ X = X + 1 = \{inl x | x \in X\} \cup \{inr()\}$

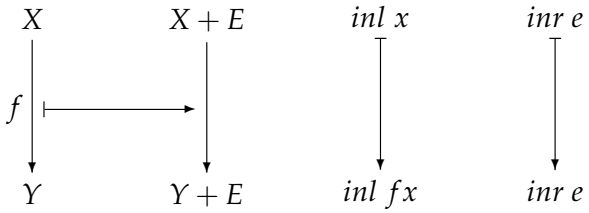
$Exc_E\ X = X + E$ where E is a set of behaviours.

The Exc (Exception monad) generalises Id (Identity monad) and $Maybe$ monad:

- Id: $E = \ = \varnothing$
- $Maybe$: $E = 1 = \{()\}$

In **Set**, where morphisms are functions, let's check that Exc_E is indeed a monad.

1. First we map functions to Exc_E:

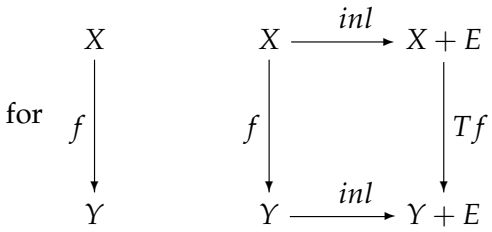


2. Unit at X (the return) is

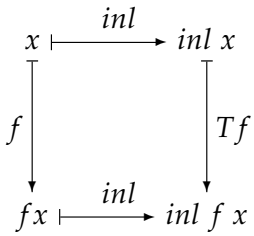
$$X \longrightarrow X + E$$

$$x \longmapsto inl\ x$$

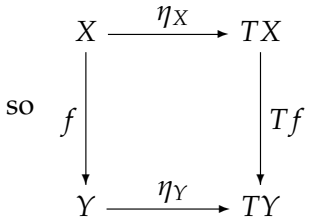
We can check that this is natural (as in definition 1):



therefore we have a single case



which does commute

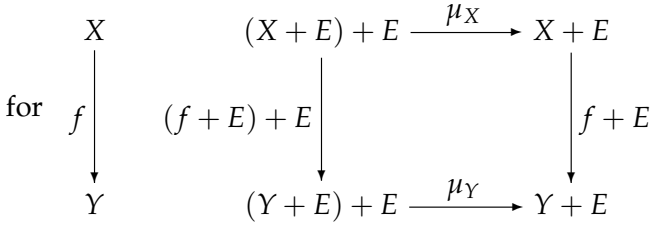


3. Multiplication $(T^2X \xrightarrow{\mu_X} TX)$ μ is:

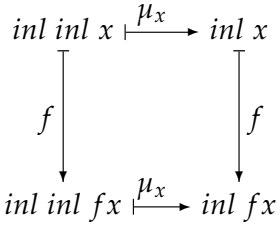
$\mu_X : (X + E) + E \to X + E$

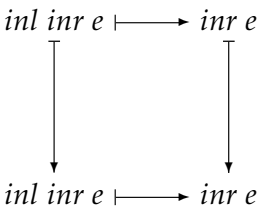
$$inl\ inl\ x \longmapsto inl\ x$$
$$inl\ inr\ e \longmapsto inr\ e$$
$$inr\ e \longmapsto inr\ e$$

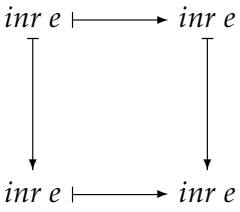
Once again, we can check it is natural:



we have to check 3 cases:

(a)


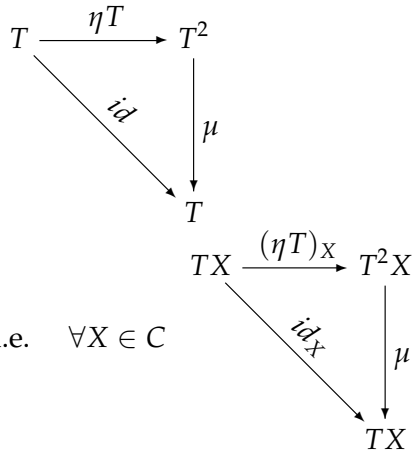
(b)


(c)


which all commute.

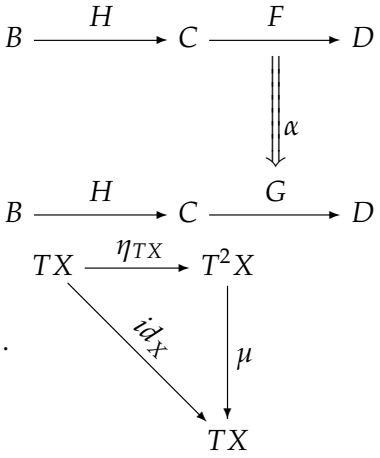
4. Finally, we have to check the three properties:

   (a) left identity law:

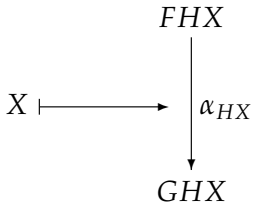


i.e. $\forall X \in C$

   This can be defined using left-whiskering:

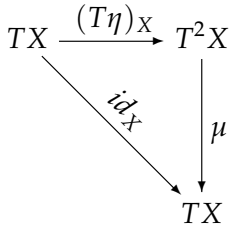


therefore

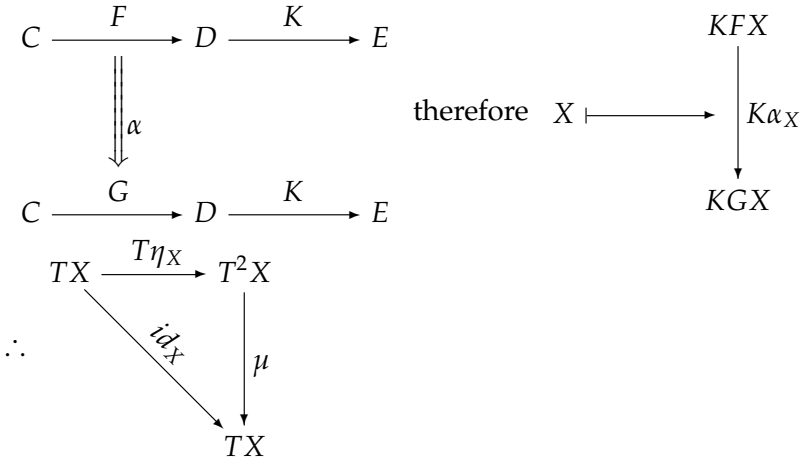


$\therefore$

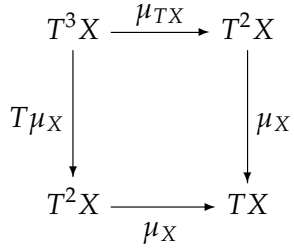
   (b) right identity law:



This can be defined using right-whiskering:

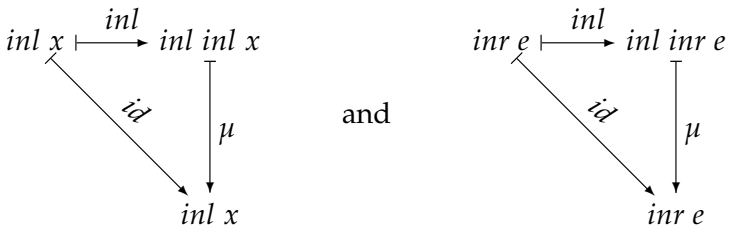


(c) associativity law:

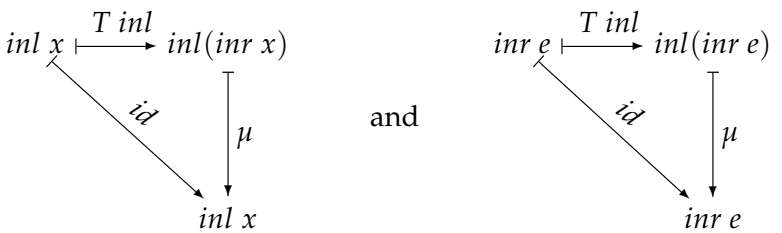


Now the actual proof that the properties indeed hold:

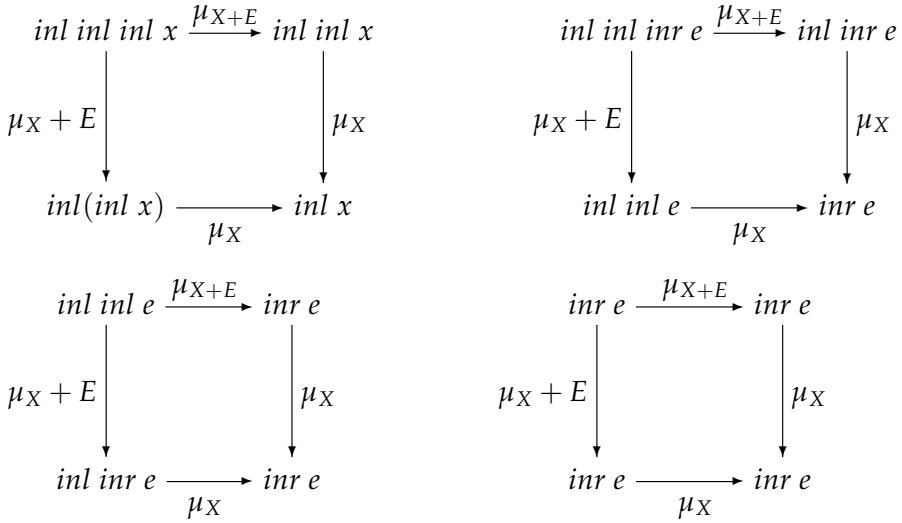
(a) There are two cases to show the left-identity law; *inl x* and *inr e*:



(b) There are two cases to show the right-identity law; *inl x* and *inr e*:



(c) There are four cases to show the associativity law; *inl inl inl x*, *inl inl inr e*, *inl inr e*, and *in e*:



With this, we have shown that the properties hold, and that Exc_E is indeed a monad.

7.4 Kleisli extension

Monads are many times implemented using the Kleisli extension. As with Haskell.

$(-)^* : hom(X, TY) \to hom(TX, TY)$

Given a monad $< T, \eta, \mu >$ over category C and a morphism $f : X \to TY$:

$f^* : TX \to TY$
$f^* = \mu_Y \circ Tf$

In other words:

$f : X \to TY$ is any function that gives us the monad type in Haskell. This also matches η, which is equivalent to the `return` function.

$f^* : TX \to TY$ is analogous to the bind function in Haskell. The difference is that arguments are in different order.

For example, if we apply this to the Exc_E monad:

$X \xrightarrow{f} Y + E$
$X + E \xrightarrow{f^*} Y + E$

so

$inl\ x \mapsto f\ x$
$inr\ e \mapsto inr\ e$

7.5 Comparison with Haskell

Monads can be defined in the standard mathematical way or through the Kleisli triple. We can either have:

- $< T, \eta, \mu >$, where T gives us the type of the monad, η is the `return` and μ is the `join`. Note that T is a functor, so in Haskell, this would be a pair $(T, fmap)$, which maps objects and functions.
- $< T, \eta, (-)^* >$, where T gives us the type, again a pair with `fmap`, η is the `return`, and $(-)^*$ is equivalent to `bind` (»=).

This can be seen more clearly with the types:

`join` and μ (multiplication):

```
join ::  T T A -> T A
```

$\mu : TTA \to TA$

`bind` and $(-)^*$ (Kleisli extension operator):

```
»= ::  T A -> (A -> T B) -> T B
```

$(-)^* : (A \to TB) \to (TA \to TB)$

We can show that a Kleisli triple on C is equivalent to the standard mathematical definition of a monad on C because:

- μ and η give you »=, i.e. $(-)^*$
- $(-)^*$ and η give you `join`, i.e. μ

7.6 Exercise: State monad

Let S be a set (for some state, i.e. the type of the state/store).

We can define a monad on **Set** with $T\ A = S \to (S \times A)$

In programming, $T\ A$ would be a stateful program that returns a value of type A.

For instance, if $A = \mathbb{N}$, then $T\ \mathbb{N} = S \to (S \times \mathbb{N})$.

We could write a program that uses this state:

$\lambda b : \texttt{bool}.(\texttt{not}\ b, b) : T\ \texttt{bool}$

This program would take the original state, invert it, and then output the original state.

With that context, we can define the monad either the standard way, or as a Kleisli triple. Defining it as a Kleisli triple:

Let *State* be the monad:

- $State\ X = S \to (S \times X)$
- The Unit (η)
  $\eta : X \to (S \to (S \times X))$
  $\eta = x \mapsto (s \mapsto (s, x))$ where $s : S$ and $x : X$
- The Kleisli operator (equivalent to bind):
  Given $g : X \to (S \to (S \times Y))$
  $g^* : (S \to (S \times X)) \to (S \to (S \times Y))$
  $g* = f \mapsto s \mapsto (g(\pi_2\ f\ s))\ (\pi_1\ f\ s)$

This is analogous to the Haskell state monad. Given *State* is analogous to our functor T:

```
newtype State s a = State  runState ::  s -> (a, s)

return ::  a -> State s a
return x = State ( \s -> (x, s) )

(»=) ::  State s a -> (a -> State s b) -> State s b
(State h) »= f
= State ( \s -> let (a,new_sate) = h s in f a new_state )
```

Alternatively:

```
type State s a = s → (a,s)
return x = λ s → (x,s)
f »= g   = λ s → case f s of (x,s') → g x s'
```

Chapter 8

Categories in Agda

8.1 Small categories

To define a small category, we will need the following definition:

- Ob : Set — Set of all objects
- Hom : Ob → Ob → Set — Set of all morphisms
- id : ∀ X : Ob .Hom X X — Identity morphism
- _;_ : ∀ X, Y, Z : Ob . Hom X Y → Hom Y Z → Hom X Z — Composition of morphisms

and the following laws:

- left-identity law
- right-identity law
- associativity law

This can be implemented in Agda:

```
record Cat : Set₁ where
  field Ob  : Set
        Hom : Ob → Ob → Set
        id  : {X : Ob} → Hom X X
        _▶_ : {X Y Z : Ob} → Hom X Y → Hom Y Z → Hom X Z

        left_id  : {X Y : Ob} {f : Hom X Y} → id ▶ f ≡ f
        right_id : {X Y : Ob} {f : Hom X Y} → f ▶ id ≡ f
        assoc    : {W X Y Z : Ob}
                   {f : Hom W X}
                   {g : Hom X Y}
                   {h : Hom Y Z} → f ▶ (g ▶ h) ≡ (f ▶ g) ▶ h
```

With this, an instance of any small category can be made.

For instance, the category of endofunctions on Booleans.

This category is one which rises from a monoid, so we use the same definition previously seen to make this category in Agda.

First, we define the data types:

```
data Bool : Set where
  true  : Bool
  false : Bool

data Unit : Set where
  unit : Unit
```

Unit is the singleton set I will be using as the objects of the category of endofunctions on Booleans. Bool is the set of Booleans which I will be using to define all endofunctions.

With this in place, we can define the category:

```
setℬ : Cat
setℬ = record
        { Ob  = Unit
        ; Hom = λ unit unit → Bool → Bool
        ; id  = λ b → b
        ; _▶_ = λ f g → λ x → g (f x)
        ; left_id  = refl
        ; right_id = refl
        ; assoc    = refl
        }
```

8.2 Set in Agda

With the previous definition, we were unable to define **Set**, however. To do this, we must define a universe level polymorphic record.

```
record Category {l : Level} : Set (lsuc l) where
  field Ob  : Set l
        Hom : Ob → Ob → Set i
        id  : {X : Ob} → Hom X X
        _▶_ : {X Y Z : Ob} → Hom X Y → Hom Y Z → Hom X Z

        left_id  : {X Y : Ob} {f : Hom X Y} → id ▶ f ≡ f
        right_id : {X Y : Ob} {f : Hom X Y} → f ▶ id ≡ f
        assoc    : {W X Y Z : Ob}
                   {f : Hom W X}
                   {g : Hom X Y}
                   {h : Hom Y Z} → f ▶ (g ▶ h) ≡ (f ▶ g) ▶ h
```

With this definition, we can finally define **Set**:

```
set : Category
set = record
        { Ob  = Set
        ; Hom = λ X Y → Set → Set
        ; id  = λ X → X
        ; _▶_ = λ {X Y Z : Set} f g → λ x → g (f x)
        ; left_id  = refl
        ; right_id = refl
        ; assoc    = refl
        }
```

8.3 Defining monads

It is possible to define a monad in the traditional way:

- T — an endofunctor on C
- η — a unit natural transformation
- id μ — a multiplication natural transformation

The downside is that it would also require the definition of a functor and all the proves associated. Alternatively, a monad can also be defined as a Kleisli Triple:

- T : Ob C → Ob C — a morphism in C
- η : X : Ob C → Hom_C(X,T X) — a unit transformation
- $(\_)^*$: X Y : Ob C → Hom_C(X,T Y) → Hom_C(T X, T Y) — the Kleisli operator

This is easier to implement since no definition of a functor is required, and no mention of naturality is given either. Hence I implemented monads as a Kleisli triple.

Given the following projections:

```
Ob      = Category.Ob
Hom     = Category.Hom
id      = Category.id
compose = Category._▶_
```

The definition of a Kleisli Triple is given as follows:

```
record Monad {l : Level}{C : Category {l}} : Set (lsuc l) where
  field T  : Ob C → Ob C
        η  : {X : Ob C} → Hom C X (T X)
        _* : {X Y : Ob C} → Hom C X (T Y) → Hom C (T X) (T Y)

        monad_lid   : {X Y : Ob C}
                      {f : Hom C X (T Y)} →
                      compose C η (f *) ≡ f
        monad_rid   : {X : Ob C} →
                      (η {X}) * ≡ id C {T X}
        monad_assoc : {X Y Z : Ob C}
                      {f : Hom C X (T Y)}
                      {g : Hom C Y (T Z)} →
                      compose C (f *) (g *) ≡ (compose C f (g *)) *
```

8.4 State monad

Now that monads are defined, it is possible to check the answer to the exercise at the end of the last chapter. We can show—in Agda—that the state monad defined previously is indeed a valid monad.

First, we need products defined:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

π₁ : {A B : Set} → (A × B) → A
π₁ (a , b) = a

π₂ : {A B : Set} → (A × B) → B
π₂ (a , b) = b

πid : {A B : Set}{p : A × B} → π₁ p , π₂ p ≡ p
πid {A}{B}{a , b} = refl
```

Assuming function extensionality:

```
postulate exten : {X Y : Set}{f g : X → Y} →
                  ((x : X) → f x ≡ g x) → (f ≡ g)

exten2 : {X Y Z : Set}{f g : X → Y → Z} →
         ((x : X)(y : Y) → f x y ≡ g x y) → (f ≡ g)
exten2 h = exten (λ x → exten(h x))
```

We can now define the state monad—on **Set**—and prove its laws:

```
state : Set → Monad {lsuc lzero}{set}
state S = record
          { T  = λ A → S → (S × A)
          ; η = λ a → λ s → (s , a)
          ; _⋆ = λ g f → λ s → g (π₂ (f s)) (π₁ (f s))
          ; monad_lid   = refl
          ; monad_rid   = lemma_rid
          ; monad_assoc = refl
          }
          where
            lemma_rid : {A : Ob set} →
                        (λf s → π₁ (f s) , π₂ (f s))
                        ≡ id set {S → (S × A)}
            lemma_rid {A} =
              begin
                (λ f s → π₁ (f s) , π₂ (f s))
                ≡⟨ exten2 (λ f s → πid {p = f s}) ⟩
                (λ f s → f s)
                ≡⟨ refl ⟩
                (λ f → f)
              ■
```