>>=

```
(>>=) :: Monad f => f a -> (a -> f b) -> f b
x >>= h = join (fmap h x)
```

For example, with the list monad, `>>=` will have the following definition.

```
(>>=) :: [a] -> (a -> [b]) -> [b]
x >>=  h = concat (map h x)
```

Arrow

Is a homorphism between objects.

Cat

Is the category of small categories which has categories as objects all functors between these as arrows.

Catamorphisms

Catamorphisms are homorphisms from Nat, or more generally homorphisms from types defined by Haskell `data` statements.

- The catamorphisms for a type are exactly the functions that can be defined using the `fold` function for the type.

Category

1. A collection of **objects**.
2. A collection or **arrows**.
3. Operations assigning to each arrow f
   (a) An object $\mathsf{dom}(f)$ called the **domain** of f, and
   (b) An object $\mathsf{cod}(f)$ called the **codomain** of f often expressed by $f : \mathsf{dom}(f) \to \mathsf{cod}(f)$
4. An associative operator $\circ$ assigning to each pair of arrows, f and g, where $\mathsf{dom}(f) = \mathsf{cod}(g)$, a **composite** arrow
$$f \circ g : \mathsf{dom}(g) \to \mathsf{cod}(f).$$
5. For each object A, an identity arrow, $id_A : A \to A$ satisfying the law that for any arrow $f : A \to B$ as
$$id_B \circ f = f = f \circ id_A.$$

Cocone

A cocone is the dual of a cone.

- A cocone for a diagram $\mathcal{D}$ in a cateogry $\mathcal{C}$ is a $\mathcal{C}-$objects, A, together with an arrow $f_i : B_i \to A$ for each object B_i in $\mathcal{D}$, such that for any arrow $G : B_i \to B_j$ in $\mathcal{D}$ such that $f_i = f_j \circ q$.

Colimit

A colimit is the dual of a limit.

Cone

For a diagram $\mathcal{D}$ in a category $\mathcal{C}$ is a $\mathcal{C}-$objects, A, together with an arrow $f_i : A \to B_i$ for each object B_i in $\mathcal{D}$, such that for any arrow $g : B_i \to B_j$ in $\mathcal{D}$, the following holds

$$g \circ f_i = f_j$$

We will use the notation $f_i : A \to B_i$ for the cone described above, and call the cone a $\mathcal{D}$-cone to indicate that it is a cone for the diagram $\mathcal{D}$.

Constant Functor

Let $\mathcal{C}$ and $\mathcal{D}$ be categories and let A be an object of $\mathcal{D}$. A constant functor K_A is a functor $K_A : \mathcal{C} \to \mathcal{D}$ that maps every object of $\mathcal{C}$ to the object A and every arrow of $\mathcal{C}$ to id_A.

Diagram

A diagram in category $\mathcal{C}$ is a collection of objects in $\mathcal{C}$ together with some (or all or none) of the arrows between those objects.

- A diagram **commutes** if whenever there are two (or more) distinct paths through the diagram from some object A to some (possible other) object B, the composition of the arrows along the other path(s).

Dual

The dual of a category $\mathcal{C}$ is identical to $\mathcal{C}$ but all the arrows are reversed. This is sometimes denoted $\mathcal{C}^{op}$.

Endofunctor

A functor from a category $\mathcal{C}$ to the same category $\mathcal{C}$.

F-Algebra

Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An F-algebra is a pair (A, a) where A is a $\mathcal{C}-$objects and $a : FA \to A$ is a $\mathcal{C}-$arrows. The object A is called the **carrier** of the F-algebra.

Final Object

A final object (or *terminal object*) in a category $\mathcal{C}$ is an object 1 such that for any object A in $\mathcal{C}$ there is exactly one arrow $\mathcal{C}$ from A to 1.

Functors

A **functor** F from category $\mathcal{C}$ to $\mathcal{D}$ is a pair of functions.

1. $F_0 : \mathcal{C}-$objects $\to \mathcal{D}-$objects
2. $F_1 : \mathcal{C}-$arrows $\to \mathcal{D}-$arrows

such that

1. If f is an arrow in $\mathcal{C}$ and $f : A \to B$ then $F_1(f) : F_0(A) \to F_0(B)$
2. $F_1(id_a) = id_{F_0(A)}$, and
3. $F_1(f \circ g) = F_1(f) \circ F_1(g)$ whenever $f \circ g$ defined.

Functors - HASK

A typical definition of a functor in haskell:

Object Functor

```
data NAME type =
        Base | Constructor1 type | Consrtuctor2 type . . .
```

Function Functor

```
instance Functor Name where
        fmap f BASE = something
        fmap f (Constructor x) = Constructor (f x)
```

HASK

In Hask, **objects are Haskell types**, so the object part of a functor, F_0, is a function from types to types.

- Therefore, a type constructor is suitable for use as the object function of a functor. For example

$$F_0 A = [A]$$

is suitable. In this case

$$F_1 = \mathsf{map}$$

completes the definition of a functor.

Identity Functor

The identity functor, Id, is an endofunctor that maps every object to itself and every arrow to itself. The identity functor for a category is just the identity arrow for that category viewed as an object in the category Cat.

Initial Object

An initial object in a category $\mathcal{C}$ is an object 0 such that for any object A in $\mathcal{C}$ there is exactly one arrow in $\mathcal{C}$ from 0 to A.

- Since 0 is an object in $\mathcal{C}$, the only arrow from 0 to itself is the identity arrow.

Isomorphic

We say that two object A and B are isomorphic if and only if there are arrows

$$f : A \to B$$

$$g : B \to A$$

such that

$$g \circ f = id_A$$

$$f \circ g = id_B$$

- Any two initial object in a category are isomorphic. The same applies to final objects.

Limit

A limit for a diagram $\mathcal{D}$ (when one exists) is a $\mathcal{D}$-cone $f_i : A \to B_i$ with the property that for any $\mathcal{D}$-cone $\{f_i' : A' \to B_i\}$ there is exactly one arrow $A' : A' \to A$

$$insert - picture - on - page - 35$$

commutes for every object B_i in $\mathcal{D}$,

- That is, a limit is a final object in a category where the objects are the cones over a given diagram

Monads

A monad is a functor, F with two natural transformations,

```
return :: a -> F a
return : Id -> F

join :: F (F a) -> F a
join : F x F -> F
```

such that the following laws are satisfied

```
join.return = id
join.fmap return = id
join.fmap join = join.join
```

Monads - HASK

Monads in Hask are implemented as:

```
return = wrap
join = concat
fmap = map
```

Natural Transformation

Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. Let F and G be functors from $\mathcal{C}$ to $\mathcal{D}$.

We write $\tau : F \to G$ to mean that τ is a natural transformation from F to G.

A natural transformation from F to G is an assignment τ that provides, for each $\mathcal{C}-$objects A, a $\mathcal{D}-$arrows, $\tau_A : F(A) \to G(A)$ such that for any $\mathcal{C}-$arrows $f : A \to B$,

$$\tau_B \circ \mathtt{fmap_F(f)} = \mathtt{fmap_G(f)} \circ \tau_\mathtt{A}$$

Object

Is any triple (S, a, f) such that

- S is a set
- $a \in s$
- $f : S \to S$

Product

A product of objects A and B is an object $A \times B$ together with two arrows $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$,

$$insert - picture - on - page - 41$$

such that for any object $\mathcal{C}$ with arrows $a : C \to A$ and $b : C \to B$.

Small Category

Is a category where the collection of objects is a set and the collection of arrows is a set.

- Any statement that is true of an arbitrary category $\mathcal{C}$ is also true of $\mathcal{C}^{op}$.

Strict

A function is called strict if $f \perp \equiv \perp$

- For example `const 5` is not strict since `const 5` $\perp \not\equiv 5$.