Category Theory

Category theory is a generalization of set theory.

By having only a limited number of carefully chosen requirements in the definition of a category, we are able to produce a general algebraic structure with a large number of useful properties, yet permit the theory to apply to many different specific mathematical systems.

Using concepts from category theory in the design of a programming language leads to some powerful and general programming constructs. While it is possible to use the resulting constructs without knowing anything about category theory, understanding the underlying theory helps.

Definition of a Category

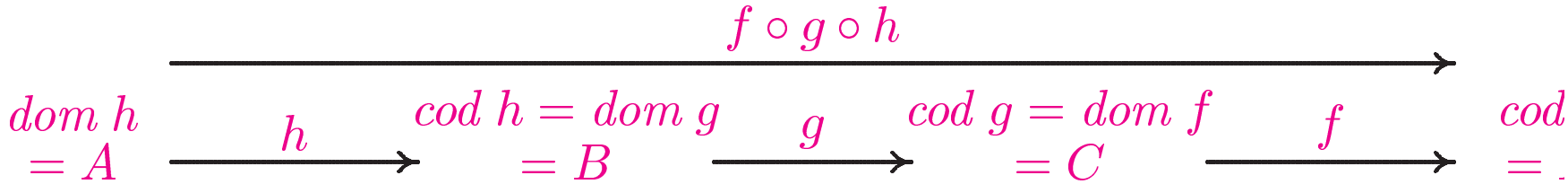
A *category* $\mathcal{C}$ is:

1. a collection of $objects$;
2. a collection of $arrows$;
3. operations assigning to each arrow f
   (a) an object $dom\ f$ called the *domain* of f, and
   (b) an object $cod\ f$ called the *codomain* of f
   often expressed by $f : dom\ f \to cod\ f$;
4. an associative composition operator $\circ$ assigning to each pair of arrows, f and g, such that $dom\ f = cod\ g$, a *composite* arrow $f \circ g : dom\ g \to cod\ f$; and
5. for each object A, an identity arrow, $id_A : A \to A$ satisfying the law that for any arrow $f : A \to B$, $\quad id_B \circ f = f = f \circ id_A$.

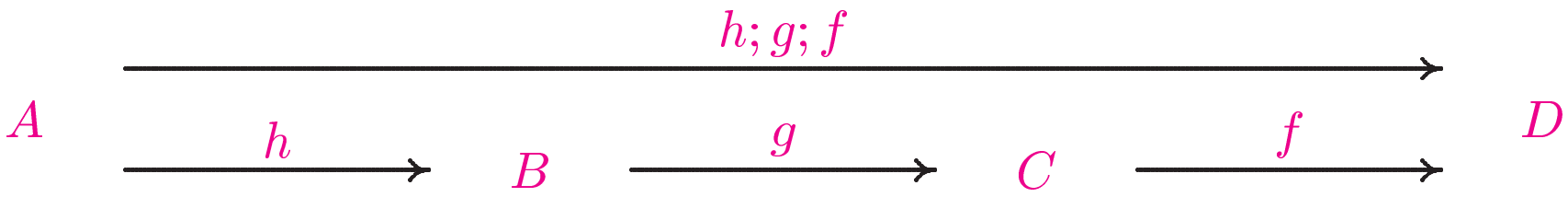
In diagrams, we will usually express $A = dom\ f$ and $B = cod\ f$ (that is $f : A \to B$) by

The associative requirement for the composition operators means that when $f \circ g$ and $g \circ h$ are both defined, then $f \circ (g \circ h) = (f \circ g) \circ h$.

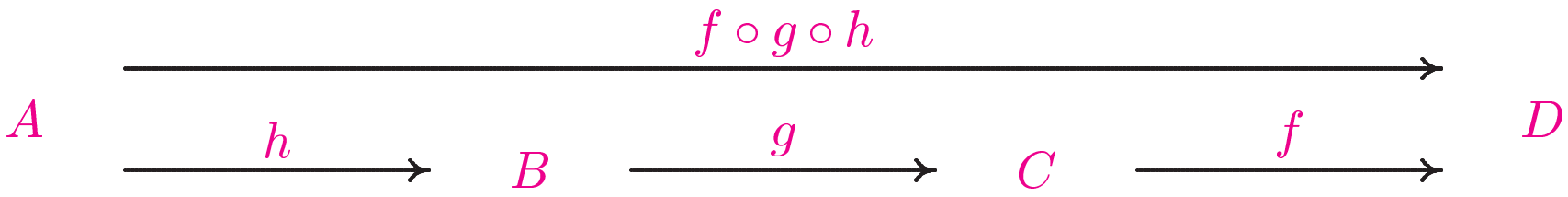
This allow us to think of arrows defined by paths throught diagrams.



I will sometimes write $(;)$ for $\texttt{flip}(\circ)$, since



is less confusing than



Examples

The category that we will use the most has all possible *Haskell* types (ground types, not polymorphic types) as objects and all possible *Haskell* (non-polymorphic) functions as arrows.

(A polymorphic datatype corresponds to an infinite collection of non-polymorphic datatypes, and similarly, a polymorphic function corresponds to the infinite collection of the possible non-polymorphic instances of the functions.)

We will call this category *Hask*. Let us complete the obvious details of the definition of $Hask$ and confirm that $Hask$ satisfies the definition of a category.

Set (*Set*): Category theory grew out of a generalization of set theory. Sets and functions between them may be viewed as a category. Objects are sets and arrows are total functions. Usually subsets are regarded as distinct sets. For example, the set of integers is not related to the set of reals. Each function is associated with a particular domain and codomain.

Discrete Categories: A discrete category is one in which the only arrows are the identity arrows.

A single set can be viewed as a discrete category, where each element of the set is an object, although this is not usually particularly useful.

The category, $\mathbf{1}$, is the one object discrete category, which is sometimes useful. (Note: Later we will use 1 to denote a terminal (or final) object in a category.)

Power Sets: A power set $\mathcal{P}(S)$ of a set S is the set of all subsets of the set S. For any set, we have a category where the objects of the category are the elements of the power set. There is at most one arrow between any two objects. There is an arrow from A to B iff $A \subseteq B$.

Any Partial Order: The power set of a set is a partial order. In a similar manner, any partial order can be view as a category.

All Partial Orders(*Poset*): There is also a category where the objects are partial orders and the arrows are monotonic functions between these partial orders.

Predicate Calculus: We can view the predicate calculus as a category, where predicates are objects and there is an arrow from A to B if and only if A implies B. Note that A implies itself, so we have identity arrows, and logical implication is transitive, so composition is well defined. There is never more than one arrow between any two predicates.

Any Group: Any group can be viewed as a category with a single object. The arrows correspond to the elements of the group. The arrow composition operator corresponds to the binary group operation.

Certain other algebraic structures can be viewed as categories in a similar manner.

All Groups (*Grp*): There is also a category where each group is an object and arrows are homomorphisms between groups.

A homomorphism is a structure preserving mapping from one set to another. For most algebraic structures, you can build categories where the arrows are homomorphisms.

For example, a homomorphism h from the group of $integers$ under $addition$, with identity element 0, to the $positive\ rationals$ under $multiplication$, with identity element 1, must satisfy the following equations.

$$h(0) = 1$$
$$h(a + b) = h(a) * h(b)$$

The exponential function (with any base) satisfies this condition, as does the constant function $h(x) = 1$.

Graphs and graph homomorphism form another category of this nature.

Functors (see the following page) are homomorphisms between categories.

Functors

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

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

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 $f \circ g$ is defined.

We write $F : \mathcal{C} \to \mathcal{D}$ to express that F is a functor from $\mathcal{C}$ to $\mathcal{D}$.

Usually we will write F for F itself, and also for both F_0 and F_1.

Functors, as defined above, are sometimes called *covariant functors* .

Once in a while, we may want a functor that reverses the direction of the arrows.

That is, we would like to change

If f is an arrow in $\mathcal{C}$ and $f : A \to B$ then $F_1(f) : F_0(A) \to F_0(B)$,

to

If f is an arrow in $\mathcal{C}$ and $f : A \to B$ then $F_1(f) : F_0(B) \to F_0(A)$,

Of course, this also means that

$F_1(f \circ g) = F_1(f) \circ F_1(g)$ whenever $f \circ g$ is defined

must be changed to

$F_1(f \circ g) = F_1(g) \circ F_1(f)$ whenever $f \circ g$ is defined.

Functors that reverse the direction of the arrows are called *contravariant functors* .

Examples

$Hask$**:** 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 = map

completes the definition of a functor.

We can think of this functor as lifting $Hask$ up to a category of list types, which is a subcategory of $Hask$.

This idea of lifting a type with a type constructor is common in *Haskell* .

Unfortunately, with this sort of functor in *Haskell* , we can't use the same name for both the function on objects and the function on arrows, as is usually done in category theory.

Instead we have a functor class

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

This allows us to associate an `fmap` function with a type constructor and use `fmap` as a generic name for the function on arrows for a type constructor viewed as the object function part of a functor.

Notice that while we have previously had classes for types (e.g. `Eq`, `Show`), `Functor` is a *constructor class* whose members are type constructors rather than types.

The *Haskell Standard Prelude* declares the list type to be a functor by

```
instance Functor [] where
   fmap = map
```

so that `fmap` can be used instead of map in contexts where is it clear that we are working with lists.

Two other functor instances are defined in the *Haskell Standard Prelude*.

```
data Maybe a =
   Nothing | Just a
     deriving (Eq, Ord, Read, Show)

instance Functor Maybe where
   fmap f Nothing = Nothing
   fmap f (Just x) = Just (f x)

instance Functor IO where
   fmap f x = x >>= (return . f)
```

We won't worry about the `IO` type until later in the course.

Quiz:

1. Find a functor $G : Hask \to \mathbf{1}$.
2. Prove that `Maybe` together with its `fmap` instance obeys the functor laws.
3. Show that for any category there exists an identity functor for that category.
4. Show that the composition of two compatible functors is a functor.

The Category of (small) Categories:

A *small category* is a category where the collection of objects is a set and the collection of arrows is a set.

The category of all small categories (*Cat*): The category Cat has all small categories as objects and all functors between these as arrows.

Of course, Cat is not a small category and therefore not an object within itself, thereby keeping Bertrand Russell happy.

Duality

Given any category $\mathcal{C}$, it is possible to produce another category, $\mathcal{C}^{\mathrm{op}}$, that is identical to $\mathcal{C}$ except that all arrows are reversed.

A quick inspection of the definition of a category should be sufficient to see that this is so.

A category $\mathcal{C}$ might seem more "reasonable" or "natural" then $\mathcal{C}^{\mathrm{op}}$. For example consider $\mathcal{S}et$. $\mathcal{S}et^{\mathrm{op}}$ is as valid as $\mathcal{S}et$ as far as the mathematics of category theory go, even though the arrows don't have a particularly nice interpretation.

We say that $\mathcal{C}^{\mathrm{op}}$ is the *dual* or *opposite* of $\mathcal{C}$.

The *Duality Principle* say that any statement that is true of an arbitrary category $\mathcal{C}$ is also true of $\mathcal{C}^{\mathrm{op}}$.

Similarly, concepts and definitions in category theory have duals.

As we will see below, an *initial object* is the dual of a *final object*. We will also see later that *sums* and *products* of objects are dual concepts.

Initial and Final Objects

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.

Similarly, 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 in $\mathcal{C}$ from A to 1.

In any category $\mathcal{C}$ we say that two objects, A and B, are isomorphic iff there exist arrows

$$f : A \to B$$
$$g : B \to A$$

such that

$$g \circ f = id_A$$
$$f \circ g = id_B$$

In the case of Set, this is just the ordinary definition of isomorphism.

A category may have no initial objects, exactly one initial object, or many initial objects. Any two initial objects in a category are isomorphic. The same applies to final objects.

Quiz:

1. Find a category with no initial objects.
2. Find a category with exactly one initial object.
3. Find a category with more than one initial object.
4. Prove that any two initial objects are isomorphic.
5. Find a category with no final objects.
6. Find a category with exactly one final object.
7. Find a category with more than one final object.
8. Prove that any two final objects are isomorphic.

Examples

In the predicate calculus, we have exactly on initial object, $False$, and one final object, $True$.

Similarly, with the power set $\mathcal{P}(S)$, the initial object is $\emptyset$ and the final object is S.

In general, a partial order may or may not have an initial object. A initial object, if it exists, is unique. Similarly for final objects.

Initial and Final Objects in Set

The initial and final objects of Set are a bit more interesting. Set has a single initial object but an infinite number of final objects.

There is exactly one initial object, $\emptyset$. For any set S, there is exactly one function, $f : \emptyset \to S$. This function is the empty function. That is, if we view a function as a set of ordered pairs, $\{(x, y)|x \in dom\ f, y = f(x)\}$, then f is the empty set of ordered pairs.

A set S is a final object in Set if and only if the set contains exactly one element.

If $S = \{s\}$, then the only function from any other set is a constant function, $f(x) = s$.

Note that there exists exactly one arrow from the initial object to any final object.

We also find in Set that the arrows going in the "wrong direction", to the initial object and from the final objects, are of interest.

In all of Set there is only one arrow to the initial object, and that is the identity arrow for $\emptyset$.

On the other hand, there are lots of arrows from the final objects. We will let 1 denote any arbitrary final object (singleton set), and S be any set. There is exactly one arrow from 1 to S for each element of S. There arrows from a final object are constants functions.

That is, since there is exactly one object in 1, each function to S takes the only element in 1 to a single element of S.

Since the arrows from 1 to S in Set correspond to the elements of S, category theory can deal with elements of a set without any need for a mechanism outside category theory.

Furthermore, this generalizes the concept of *element* to categories other than Set.

Notice that while initial objects and final objects are dual concepts, they are not dual to each other within the specific category Set.

Diagrams, Cones, Limits and their Duals

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

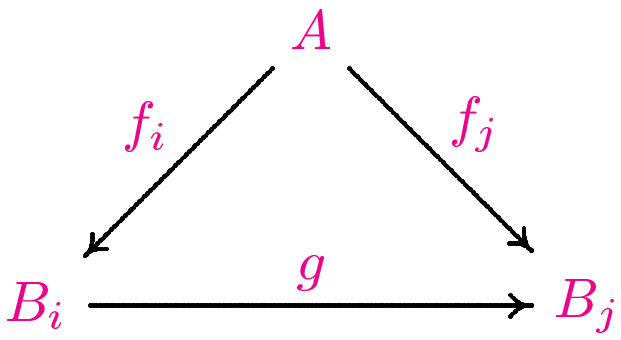
We say a diagram *commutes* if whenever there are two (or more) distinct paths through the diagram from some object A to some (possible other) object B, the composition of the arrows along one path is equal to the composition of the arrows along the other path(s).

A *cone* for a diagram D in a category $\mathcal{C}$ is a $\mathcal{C}$-object, A, together with an arrow $f_i : A \to B_i$ for each object B_i in D, such that for any arrow $g : B_i \to B_j$ in 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 *D-cone* to indicate that it is a cone for the diagram D.

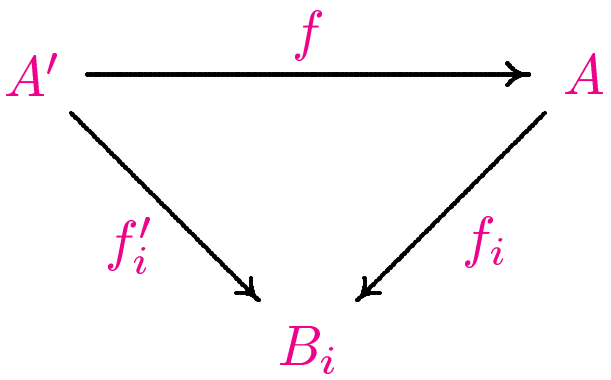
We can express the equation $g \circ f_i = f_j$ from the previous page by saying that



commutes.

The above commutative diagram is a different diagram than D but is useful for describing D.

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



commutes for every object B_i in D.

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

As a trivial example, consider a limit of an empty diagram, D, in a category $\mathcal{C}$.

Since the empty diagram has no objects or arrows, a D-cone is just an object, and every object of $\mathcal{C}$ is a D-cone.

It follows that a limit of an empty diagram is a final object, since every other D-cone has a unique arrow to this object.

Diagrams, Cones, Limits

A *cocone* is the dual of a cone. Hence:

> A *cocone* for a diagram D in a category $\mathcal{C}$ is a $\mathcal{C}$-object, A, together with an arrow $f_i : B_i \to A$ for each object B_i in D, such that for any arrow $g : B_i \to B_j$ in D, the following holds. $f_i = f_j \circ g$

A *colimit* is the dual of a limit.

Quiz:

1. What is a colimit of an empty diagram, and why.
2. Describe a limit of a diagram consisting of a single object.

Sums and Products

We have previously talked about sums (disjoint unions) and products (cartesian cross products) and their relationship to *Haskell* algebraic datatypes. We will now consider these concepts in terms of category theory.

There are three advantages to the categorical view. First, it removes the issue of representation. For example, a product can be represented in any of a number of ways. Our categorical definition of category will not specify a representation, but will instead yield all possible (isomorphic) products. Second, the concepts of sum and product will extend to categories where object need not be sets or even similar to sets. Third, we sill see that product and sum are dual concepts.

A product will be an initial object of a suitable category and a sum will be a final object of a similar category.

In order to avoid a specific representation for a product, we will actually consider a product of A and B to be an object C together with two arrows $\pi_1 : C \to A$ and $\pi_2 : C \to B$. In the case of Set, these arrows will be the projection functions (e.g. `fst` and `snd` is *Haskell*).

$$A \xleftarrow{\pi_1} C \xrightarrow{\pi_2} B$$

Similarly, a sum of A and B will be an object C together with a pair of arrows, which will be injection functions in the case of Set.

$$A \xrightarrow{i_1} C \xleftarrow{i_2} B$$

A *product* of two objects, A and B is a limit of the diagram D consisting of the two objects A and B and no arrows.

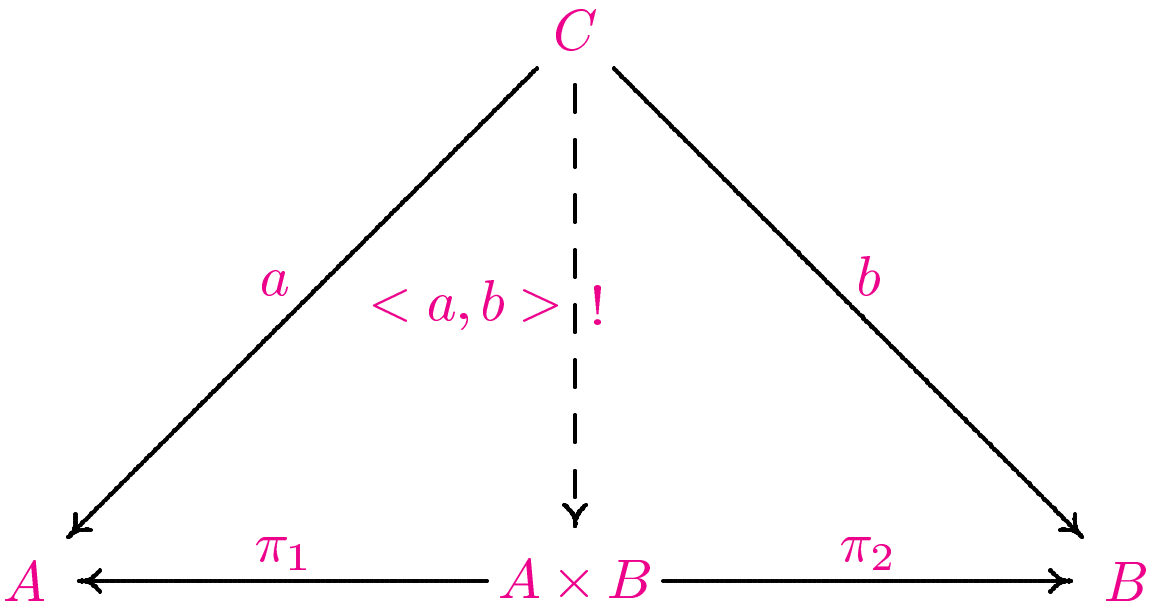
A *coproduct* (or *sum*) of two objects, A and B is a colimit of the diagram D consisting of the two objects A and B and no arrows.

A category may or may not have products and coproducts.

Let us express the definition of a produce in another way.

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 $\pi_2 : A \times B \to B$,

such that for any object C with arrows $a : C \to A$ and $b : C \to B$ (that is, any cone for the diagram consisting of A and B) there exists a unique arrow, which we will call $< a, b >$, such that the diagram on the following page commutes.



The dotted arrow asserts that the arrow must exist and the ! asserts that the arrow is unique.

The diagram on the previous page states that there must exist a unique arrow, $< a, b >$, such that

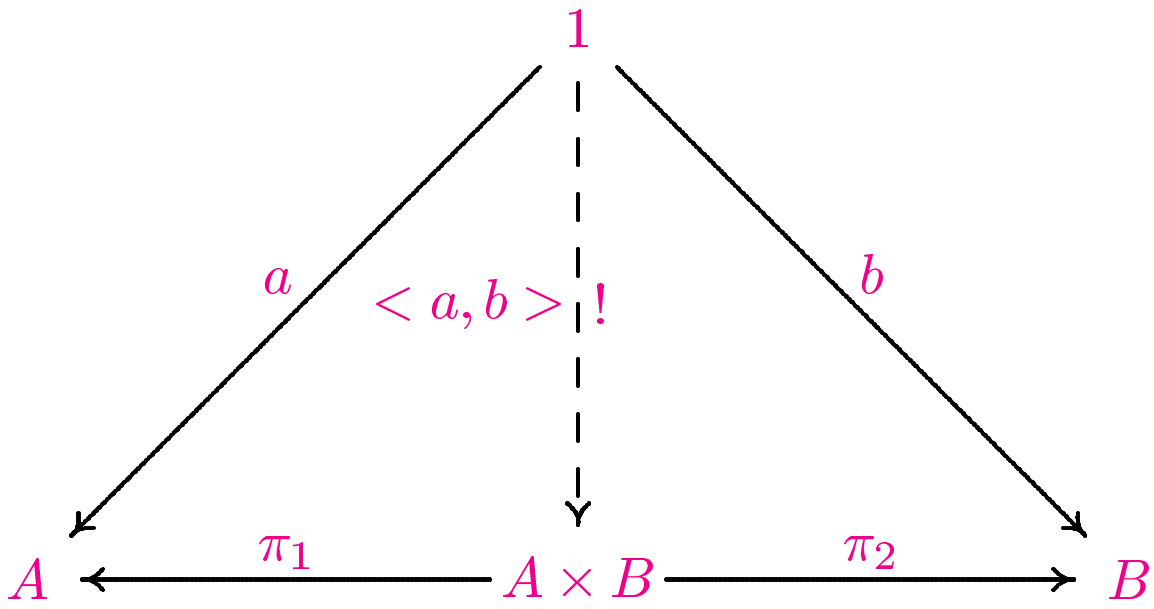
$$a = \pi_1 \circ < a, b >$$

$$b = \pi_2 \circ < a, b >$$

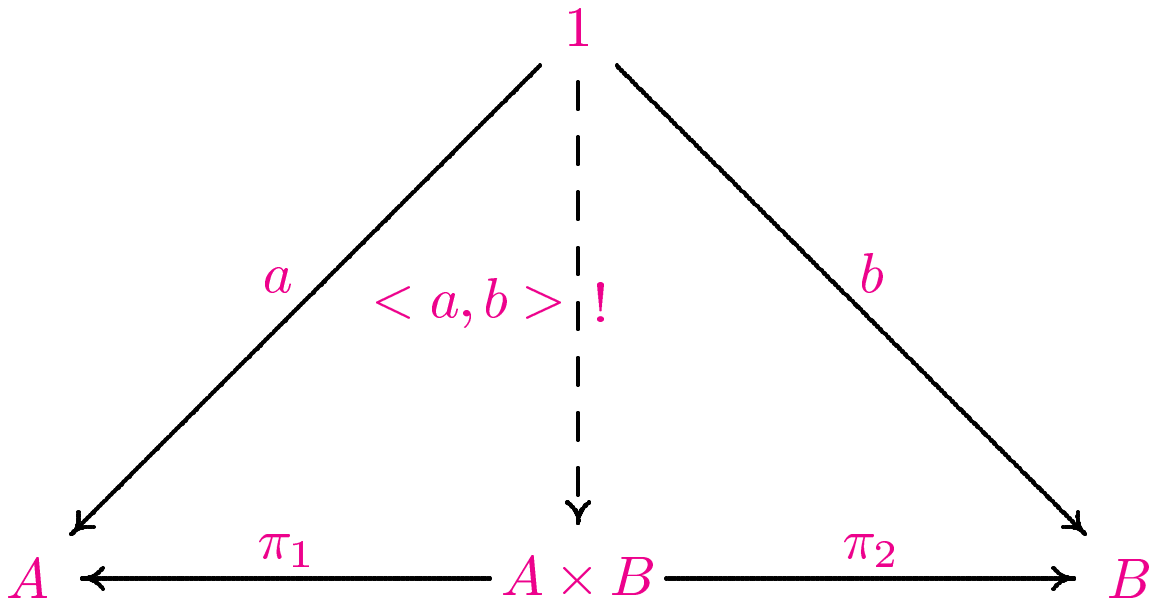
Quiz: In the category $\mathcal{S}et$, what happens if either or both sets in a product are empty? What happens if either or both are singleton sets?

Quiz: Prove that if a pair of object in any category has two (or more) distinct products then the products are isomorphic.

In Set, by considering the case where $C = 1$ (a singleton set), we can see that a "value" in $A \times B$ must be able to store any value from A and any value from B, since the arrows a and b can "select" any pair of values in A and B respectively.

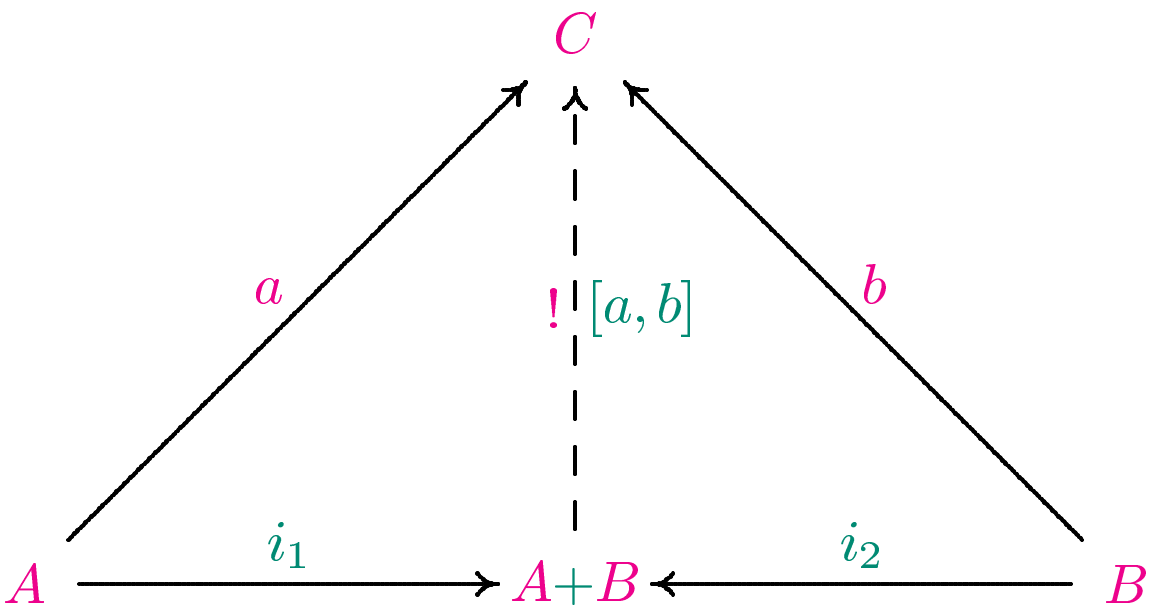


On the other hand, if $A \times B$ could store any additional information, then the arrow $< a, b >$ would not be unique.



Furthermore, while a product must contain exactly the right amount of information, there is no restriction on the way this information is "represented". Any product is as good as any other, and they are all isomorphic to each other.

A sum (or coproduct) of objects A and B is an object $A + B$ together with two arrows $i_1 : A \to A + B$ and $i_2 : B \to A + B$ such that for any object C with arrows $a : A \to C$ and $b : B \to C$ (that is, any cocone for the diagram consisting of A and B) there exists a unique arrow, which we will call $[a, b]$ such that the following diagram commutes.

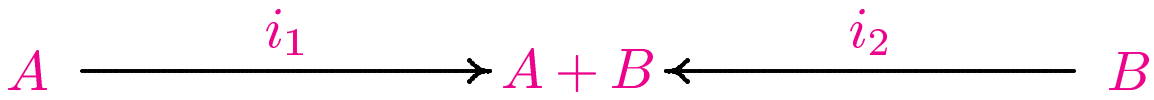


Notice that the sum figure is the same as the product figure except for the name changes, shown in PineGreen, and the direction of the arrows, all of which are reversed.

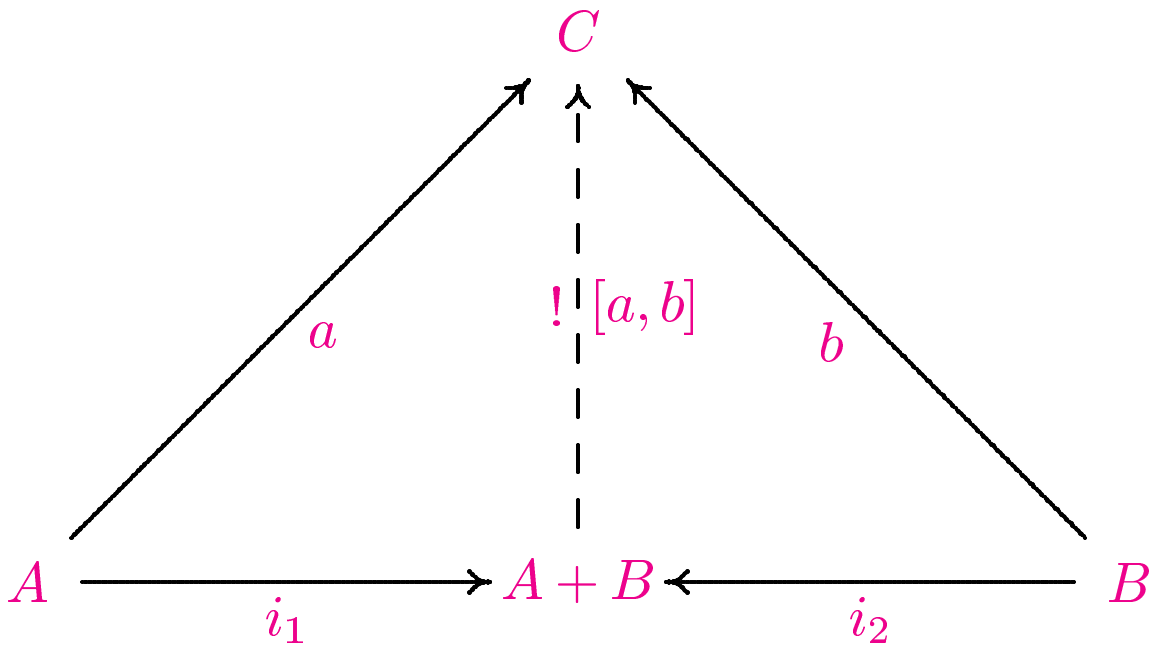
This again illustrates the duality of sums and products.

To get a better feel for sums, let's consider some special cases of sums (disjoint unions) in Set.

Since a sum of objects A and B is an object $A + B$ together with two arrows $i_1 : A \to A + B$ and $i_2 : B \to A + B$, we will consider the sum figure assuming that the bottom part of the figure,



is fixed, but that consider various possible cases for C and the arrows to C.



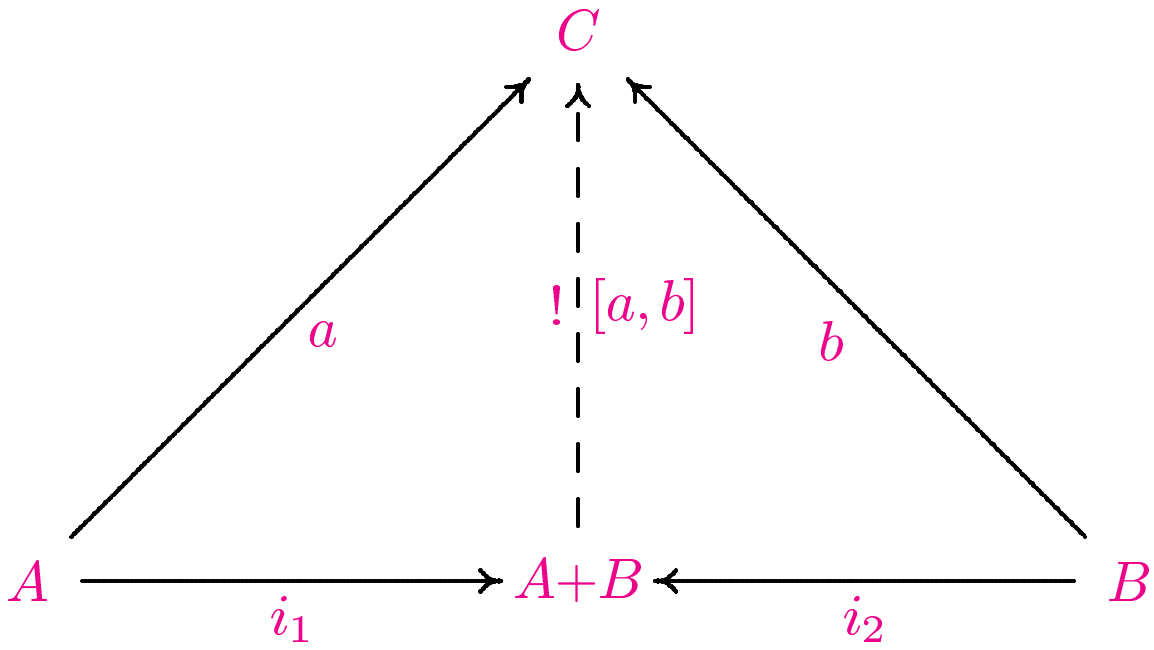
1. Suppose $C = A$ and $a = id_A$, with b being arbitrary. (The other way around is similar.)
2. Suppose A and B are nonempty and $C = \{1, 2\}$, $a = \texttt{const}\ 1$ and $b = \texttt{const}\ 2$.

1. Suppose $C = A$ and $a = id_A$, with b being arbitrary. (The other way around is similar.)

   Observation: For any $a \in A$, we can insert the value into $A + B$ and get it back out again. Similarly for any $b \in B$. Hence, at the very least, $A + B$ can store either an element of A or an element of B, as we would expect from any sort of union.

2. Suppose A and B are nonempty and $C = \{1, 2\}$, $a = \texttt{const}\ 1$ and $b = \texttt{const}\ 2$.

   Observation: This shows that we can always determine which insertion function produced the value in $A + B$. Notice that this works even in the case where $A = B$.



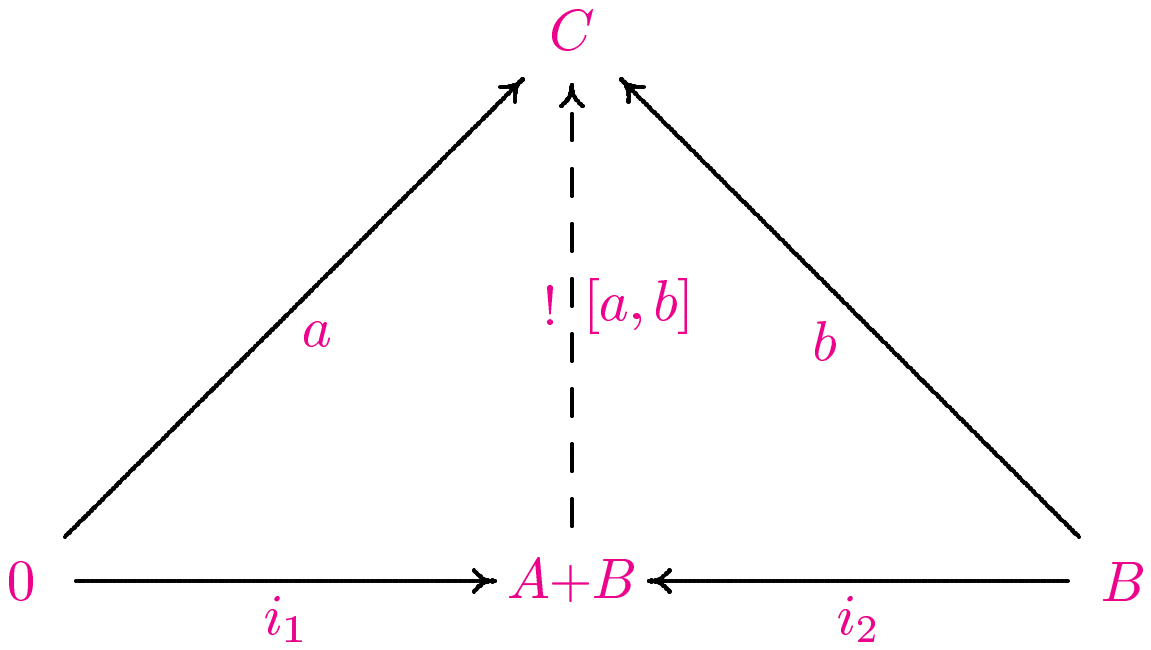
1. Suppose that $C = A + B$ and $a = i_1$ and $b = i_2$. Clearly $[a, b] = id_{A+B}$ will work. Since $[a, b]$ must be unique, nothing else can work.
2. Consider the case where $A = \emptyset$ and/or $B = \emptyset$.

1. Suppose that $C = A + B$ and $a = i_1$ and $b = i_1$. Clearly $[a, b] = id_{A+B}$ will work. Since $[a, b]$ must be unique, nothing else can work.

   Observation: $A + B$ can't contain any more than just the necessary information, since there is no extra information that can be changed in the arrow from $A + B$ back to itself. If $A + B$ contained some extra junk, then we could get additional arrows from $A + B$ back to itself that would allow the diagram to commute.

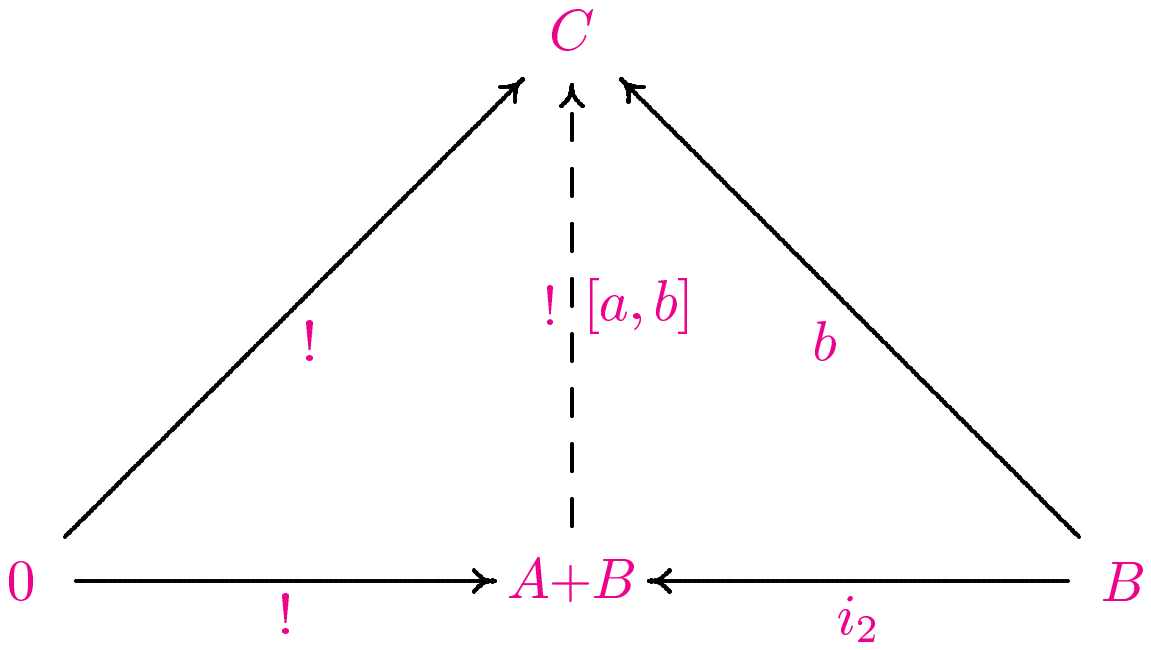
2. Consider the case where $A = \emptyset$ and/or $B = \emptyset$.

   Observation: Let us do a general category theoretic proof here, so we can apply the result more generally.



By definition, there is exactly one arrow from 0 to any object. Therefore the left triangle must commute. That is, $i_1; [a, b] = a$ because $i_1; [a, b]$ and a are both the (same) unique arrow from 0 to C.

Therefore, a possible sum is $A + B = B$ so $[a, b] = b$ for any C and b. Any other sum must be isomorphic. (This is the dual of a property for products that you proved in a quiz.)



It follows that $0 + B \cong B$ for any B in any category with sums. Similarly, $A + 0 \cong A$ and $0 + 0 \cong 0$.

Since $0 + B \cong B$, by duality (reversing the arrows to produce the opposite category) $1 \times B \cong B$ in any category. This proves:

In Set this gives us the obvious results.

In the predicate calulus, this says $true \wedge B \equiv B$ and $false \vee B \equiv B$.

In powersets, where S is the universal set, we have $S \cap B = B$ and $\emptyset \cup B = B$.

With partial orders, where everything is defined, $\top \sqcap B = B$. (That is the minimum of the maximal element and B is B. For example $\infty \min B = B$ with integers — or reals.) Similarly $\bot \sqcup B = B$.

Lots of similar results in other categories also follow.

Remember that $0 \times B \cong 0$ in the category of sets.

Similar results apply in lots of other categories, for example $false \wedge B = false$, $\emptyset \cap B = \emptyset$, etc.

The duals of these last properties, $true \vee B = true$ and $S \cup B = S$ also hold.

However, in all of these cases, we have categories where there is a unique initial object (or unique final object in the dual cases), and no other object has an arrow to the unique initial object (or in the dual cases, from the unique final object).

The dual of our first result above (in the category of sets) is $1 + B \cong 1$, which is not true in the category of sets. In this category there are multiple final objects, and every object except the initial object has arrows from every final object.

Polynomial Functors, etc.

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

We may write $\mathbf{1}$ for $\mathbf{K}_A$ in the special case where A is a final object and we really don't care which final object A is.

You should confirm that a constant functor is indeed a functor.

An *endofunctor* is a functor from some category $\mathcal{C}$ to the same category, $\mathcal{C}$.

The *identity functor* , $\mathbf{Id}$, is an endfunctor 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 $\mathcal{C}at$.

Assume that all binary products exist in a category $\mathcal{C}$. That is, for any objects A and B, $A \times B$ exists.

Let F and G be any pair of endofunctors defined on $\mathcal{C}$. We define the endofunctor $(F \times G)$ by

$$(F \times G)\, A = F\, A \times G\, A$$
$$(F \times G)\, h = < F\, h \circ \pi_1, G\, h \circ \pi_2 >$$

Notice that if $h : A \to B$ then $(F \times G)\, h : (FA \times GA) \to (FB \times GB)$.

The definition of the product of functors generalizes in a straightforward manner to products of any finite degree, but we will restrict our attention to binary products in order to simplify the presentation.

Similarly, we will consider only binary coproducts (sums) of functors.

Assume that all binary coproducts (sums) exist in a category $\mathcal{C}$. That is, for any objects A and B, $A + B$ exists.

Let F and G be any pair of endofunctors defined on $\mathcal{C}$. We define the endofunctor $(F + G)$ by

$$(F + G)\, A = F\, A + G\, A$$
$$(F + G)\, h = [i_1 \circ F\, h, i_2 \circ G\, h]$$

Notice that if $h : A \to B$ then $(F + G)\, h : (FA + GA) \to (FB + GB)$.

Let's see how these ideas carries over to $Hask$ and *Haskell*.

Recall that the objects of $Hask$ are pointed cpos and each object correspond to a particular ground type (i.e. a type with no type variables) in *Haskell*. The arrows in $Hask$ are continuous functions corresponding to *Haskell* functions that are not polymorphic or overloaded.

Let's ignore the $\perp$ element that is included in every object in $Hask$, and further ignore the fact that every $Hask$ object has at least one value other than $\perp$, so $Hask$ has no initial or final objects.

To save room, we will use the symbol $\bowtie$ to mean "corresponds to".

With $Hask$ objects and *Haskell* types,

$A \times B \quad \bowtie$ `(A, B)`

$A + B \quad \bowtie$ `Either A B`

With $Hask$ arrows and *Haskell* functions,

$< f, g > \quad \bowtie$ `pair(f,g)` (See page 42 of Bird.)

$[f, g] \quad \bowtie$ `either f g` which is the same as

`case(f,g)` (See page 46 of Bird.)

With $Hask$ functor operators and *Haskell* functions, the arrow part of the functor gives

$(F \times G) \quad \bowtie$ `cross(fmap`^{F}`,fmap`^{G}`)` (See page 42 of Bird.)

$(F + G) \quad \bowtie$ `plus(fmap`^{F}`,fmap`^{G}`)` (See page 46 of Bird.)

while for the object part we have

$(F \times G)A \quad \bowtie$ `(F A, G A)`

$(F + G)A \quad \bowtie$ `Either (F A) (G A)`

Recall Bird's comment, in ¶2 of page 47, that "The algebraic properties of `case` and `plus` are dual to those of `pair` and `cross`."

Let's consider a full *Haskell* datatype declaration.

```
data Perhaps a = Nope | Only a | Both Int a
```

We have

$$\texttt{Perhaps} \bowtie (\mathbf{1} + Id + (\mathbf{K}_{\texttt{Int}} \times Id))_{\perp}$$

and could make `Perhaps` an instance of `Functor` by

```
instance Functor Perhaps where
   fmap f Nope = Nope
   fmap f (Only a) = Only (f a)
   fmap f (Both n a) = Both n (f a)
```

Notice:

1. We used "$|$" directly rather than `Either`, in part because we have a sum of degree three instead of degree two, and in part because `Either` is just a way to encapsulate "$|$" and supply data constructors (tags) `Left` and `Right`.

2. In *Haskell*, `Nope`, `Only` and `Both` are names of constructors (instead of i_1 and i_1, or `Left` and `Right`) as well as tags used in pattern matching. These names have no role in the structure of the functor.

3. We have added the $\perp$, whose existence we now acknowledge. We will add a $\perp$ to the functor corresponding to the right side of each *Haskell* `data` statement.

A *polynomial functor* is any functor formed using constant functors, Id and $+$ and $\times$.

In *Haskell* we need to be able to lift functors, using $\perp$. This is done in the obvious way and we will skip the details.

Haskell also allows function types to be used in datatype declarations. While category theory supports this, we will skip the details, since lifted polynomial functors are sufficient for illustrating the ideas that we will be exploring.

F-Algebras

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 $\mathcal{C}$-object and $a : F\,A \to A$ is a $\mathcal{C}$-arrow. The object A is called the *carrier* of the F-algebra.

For example, let $Natural$ be the set of natural numbers in the category Set. We will now define an F-algebra on the carrier $Natural$ with two "operators". The first operator is really a constant, which can be viewed as an operator that takes zero arguments. This operator, which we will call $zero$, "returns" the natural number 0. The other operator is a binary operator, $plus$, that returns the sum of its two arguments. This gives us a group with an identity element and a binary operation.

This example could easily be extended to include more constants and operators, such as the constant one and the binary operator $times$, as well up operators of different arity, such as a unary operator, $negate$, and operators of arity three or more.

Let $F : F\ Set \rightarrow Set$ be the functor defined by $F_0\ S = 1 + S \times S$ and $F_1\ f = [id_1, < f, f >]$.

We now define the arrow $a : F\ A \rightarrow A$ as follows.

$$a(i_1(x)) = 0$$
$$a(i_2(a, b)) = a + b$$

Let us revisit this example in *Haskell* .

We will need to assume that we have a *Haskell* type `Natural` that contains the natural numbers and nothing else. This is not actually possible in *Haskell*, because every type contains a $\perp$ value, but we can pretend. Similarly, for a later example, we will assume that *Haskell* contains a type `PosReal`, which includes all positive real numbers (so it has an uncountable number of values), does not contain $\perp$. We will distinguish that fake types having no $\perp$ element by using a `special color (PineGreen)`.

We start by defining the functor (type constructor) `F`.

```
datatype F a = Ident | Op a a

instance Functor F where
   fmap f Ident = Ident
   fmap f (Op x y) = Op (f x) (f y)
```

As the color suggests, we assume that no $\perp$ element is added by the type constructor `F`. We also create labels, `Zero` and `Add` for the two cases in the domain sum.

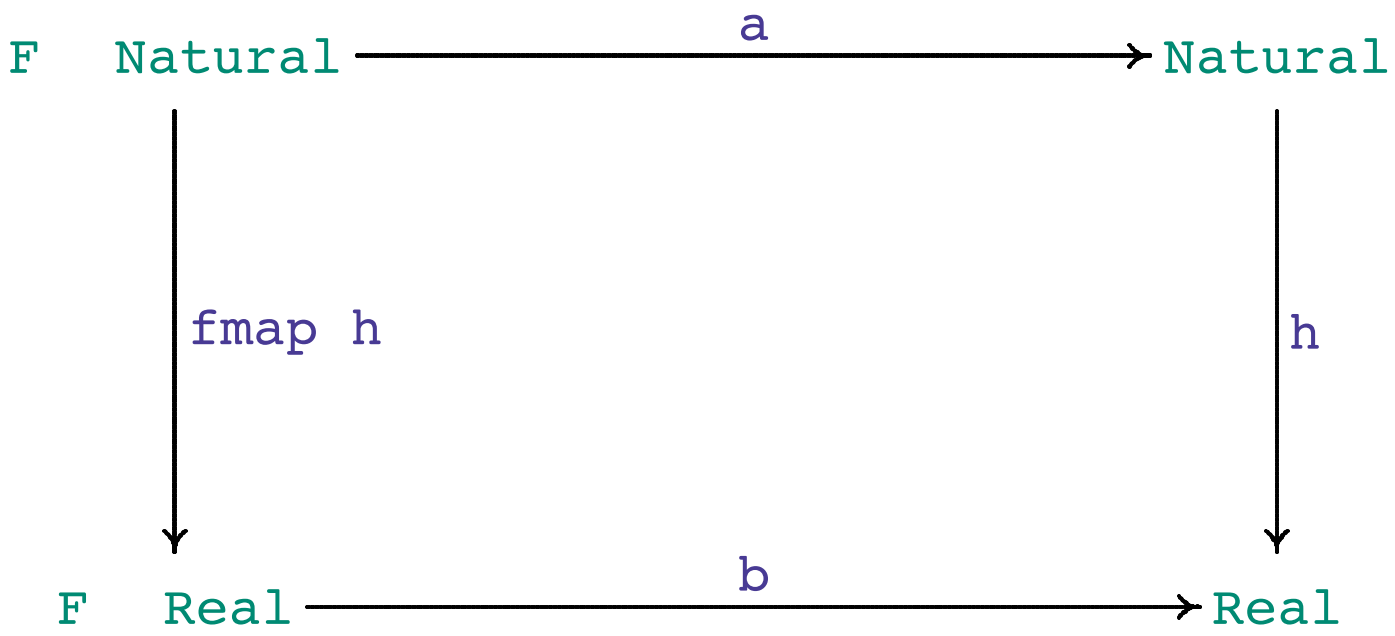
We now define

```
a :: F Natural -> Natural
a Ident = 0
a (Op x y) = x + y
```

Notice how a single function, `a`, defined both (or in general case, all) operations at once.

Using the same functor, `F`, we also can define an F-algebra over the positive real numbers with a constant *one* and an operation *times*.

```
a :: F PosReal -> PosReal
a Ident = 1
a (Op x y) = x * y
```


F Natural
a
Natural
fmap h
h
F Real
b
Real

The *Natural Numbers* as an Initial Object

Consider the *Haskell* datatype declaration:

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

This suggests that `Nat` corresponds to a domain Nat that satisfies the following isomorphism.

$$Nat \cong (\mathbf{1} + Nat)_\perp$$

If we define a type constructor, `NatF`, which we will view as a functor, by

```
data NatF nat = Zero | Succ nat
```

then we will take `Nat` to be isomorphic to a type `Nat2` such that

```
Nat2 ≅ NatF Nat2
```

```
newtype Mu f = In f (Mu f))
```

Initial and Final Objects in Grp

The category Grp has a final object, a single element group, but no initial object.