Intro to Category Theory: Functors

1 Functors

Here's my slightly cheeky intro to functors. Category theory is about studying structure preserving transformations (morphisms). Naturally, to understand categories we should understand the structure preserving morphisms between categories. These structure preserving transformations are indeed functors. I don't think that's a particularly good motivation. More practically, pretty much all future definitions are going to require functors. Another benefit is that functors are a concept in functional programming.

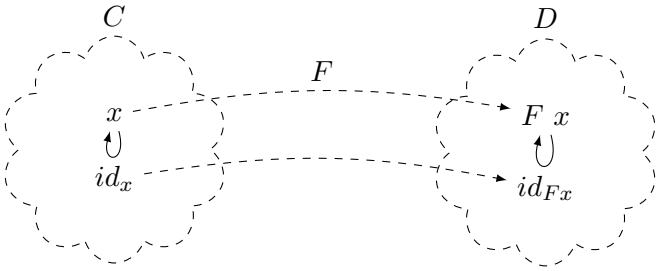
Functors are morphisms for categories. That is, functors are going to take us from one category to another, while preserving the axioms of the first category. Categories have three axioms: unit, composition, and associativity. Associativity is handled because the target of the functor is a category, but the other two are not guaranteed. Thus, they are part of the definition.

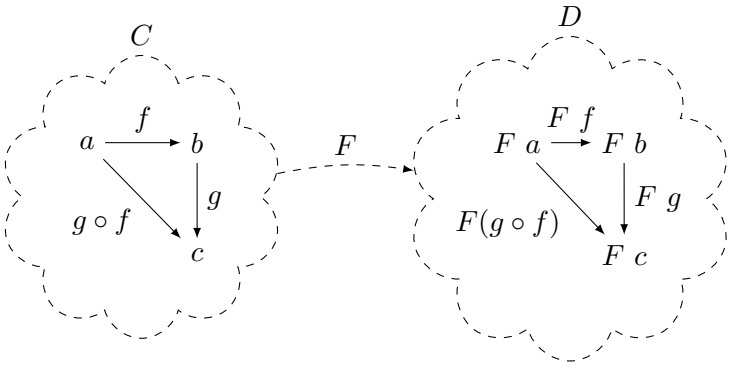
Definition 1.1. A *functor* F from category C to category D ($F : C \to D$) maps objects of C to objects of D, denoted $F\ c$ for an object c of C, and morphisms $f : a \to b$ of C to morphisms of D, denoted $F\ f : F\ a \to F\ b$, such that

- (Unit) $F\ id_x = id_{Fx}$
- (Composition) Let $f : a \to b$ and $g : b \to c$ be morphisms of C, then $F\ (g \circ_C f) = (F\ g) \circ_D (F\ f)$.

A couple points about the definition. For composition, I have subscripted the composition operator $\circ$ with the category that the composition exists in. This subscript is never written. I have just done so to indicate that we have two different compositions that is related through F. I will not use a subscript from here on out. Another slightly annoying piece but standard notation is that functors map objects and morphisms using the same notation that is $F\ a$ is an object of D and $F\ f$ is a morphism of D; however, you can only know this if a has been introduced as an object and f a morphism.

Here are some diagrams to illustrate the pieces of the definition. These diagrams are not meant to be taken as replacements for the definition.





One of the nice things about the definition of a functor is that they preserve commutative diagrams.

Theorem 1.1. *Functors preserve commutative diagrams. That is, consider a functor $F : C \to D$. If there is a commutative diagram in C the image of that diagram through F must also be a commutative diagram in D.*

Proof. This isn't a totally rigorous proof, but the idea is that the composition requirement of a functor ensures commutation holds. Let $< f_1, ..., f_k >$ and $< g_1, ..., g_l >$ be two paths of a commutative diagram of C that have the same start and end points. Then,

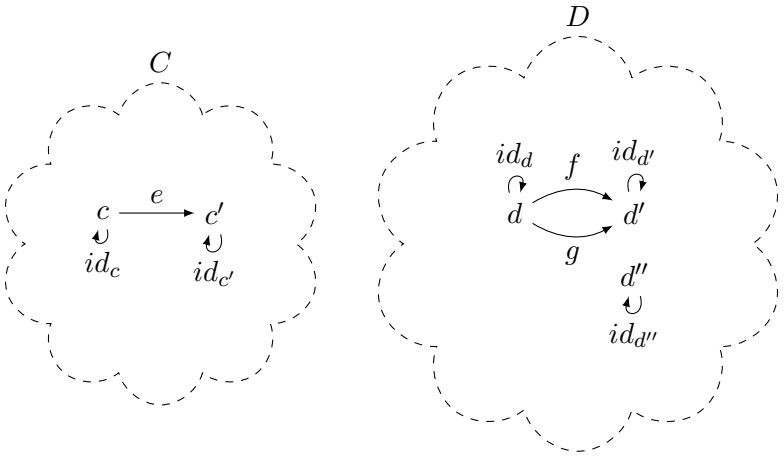
$$(F\ f_1) \circ ... \circ (F\ f_k) = F(f_1 \circ ... \circ f_k) = F(g_1 \circ ... \circ g_l) = (F\ g_1) \circ ... \circ (F\ g_l)$$

□

I want to make a point about intuition. As a computer scientist, I tend to think of functions as being or representing some sort of process. $f(x) = x + 2$ intuitively means to take x and add 2 to it. When we learn about functions in elementary school we assign some computational element to them. This is not really correct in the mathematical sense. Mathematical functions are simply sets of pairs. They are really more of assignments rather than computations. They always terminate for example. There's a huge literature in programming languages about the difference between mathematical functions and computations, and I don't mean to open this can of worms. However, I do want to make sure you have a more restrained intuition about functors. Functors are essentially (though not exactly) functions for categories. I don't want you to think of functors as having a computational element. It's not always wrong, but categories and functors are very abstract and their utility comes more as being an assignment from one category into a piece of another. Thus, I think you should think of functors as "embedding" one category into another. Some other language that gets used a lot is thinking of a functor as a "box" or a "container". Though this viewpoint isn't wholly uncontroversial I think it is a good idea to think of a functor as a container or an embedding.

1.1 Toy Example

Here's a little toy example of a functor:



Example 1.1.
Let $F : C \to D$ be a functor with the following mappings:

$$F\ c = d \qquad F\ e = f$$
$$F\ c' = d' \qquad F\ id_c = id_d$$
$$F\ id_{c'} = id_{d'}$$

This wasn't a particularly interesting example, but it does show how F embeds C into D.

2 Functors in Programming

Functors are a defined construct in many functional programming languages. While there are functors in ml languages, those are not exactly functors in the categorical sense. Haskell on the other hand has a defined functor typeclass which implements the category theory functor. It turns out that many normal, non-scary data types are actually functors. Here we will look at two examples.

First, let's get a little perspective. We are operating in the category **Hask** of types and functions. Our functors will take in a type and produce another type and take a function and produce another function. That is, our functors will be from **Hask** to **Hask**. Functors which have the same source and target are called *endofunctors*.

Here is the typeclass declaration for a functor in haskell:

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

In this case the declaration is stating that there is a type constructor `f` called a `Functor`, which has a function `fmap` that takes in a function and produces another function. We can think of `f` being a container, and `fmap` does the job of taking a function from type `a` to `b` and producing a function from a container of `a` to a container of `b`.

2.1 Maybe Functor

A concrete example is the `Maybe` type constructor. `Maybe` in other languages is also call option, but the idea is to have a type represent failing computations. Heres the declaration for the `Maybe` constructor and the definition for `fmap`:

```
data Maybe a = Nothing | Just a

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

We'll return to the `Maybe` functor when we talk about monads.

2.2 List Functor

A more familiar example of a functor is a list type

```
data List a = Nil | Cons a (List a)

instance Functor List where
        fmap f Nil = Nil
        fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

There's a lot more example of functors in programming, and we will be returning to their uses and furthering these examples later. An important note about the haskell compiler. The categorical requirements of a functor (unit and composition) are not guaranteed to be automatically satisfied. It is on the programmer to ensure that their implementations satisfy the functor laws. It's a good exercise to check to see if the above implementations satisfy the functor laws. That is check that `fmap id = id` and `fmap (g.f) = (fmap g). (fmap f)` for all relevant `f` and `g`.

3 Some Functor Terminology

A trivial, but useful, functor in category theory is the *constant* functor.

Definition 3.1. The *contant functor* $\Delta_d : C \to D$ is defined as follows:

- $\Delta_d\ c = d$ for all objects c of C.
- $\Delta_d\ f = id_d$ for all morphisms f of C.

The constant functor collapses an entire category down to one object.
As stated before a functor which has the same source and target is call an endofunctor.

Definition 3.2. A functor $F : C \to C$ is called an *endofunctor*.

There's a particular endofunctor for each category called the identity functor.

Definition 3.3. The *identity functor*, id_C or $1_C : C \to C$ is defined as $id_C\ c = c$ and $id_C\ f = f$ for all objects and morphisms respectively.

With the identity functor defined we can define a new category

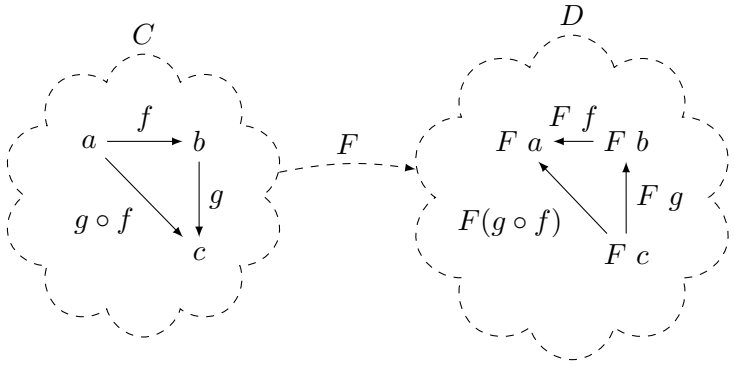
Definition 3.4. The category of all (small) categories, **Cat**, has objects all small categories, morphisms functors, composition is functor application, and identity morphisms are identity functors.

There's some more functor terminology which we have to talk about. There are actually two types of functors: *covariant* and *contravariant*. Covariant functors are the ones we've been talking about. They are "normal" functors. When a covariant functor operates on a morphism the direction of the arrow is preserved in the target category. However, the direction of a morphism flips in the case of a contravariant functor.

Definition 3.5. A *contravariant* functor $F : C \to D$ maps objects of C to objects of D, and maps a morphism $f : a \to b$ of C to a morphism $F\ f : F\ b \to F\ a$, such that

- (Unit) $F\ id_x = id_{Fx}$
- (Composition) Let $f : a \to b$ and $g : b \to c$ be morphisms of C, then $F\ (g \circ f) = (F\ f) \circ (F\ g)$.

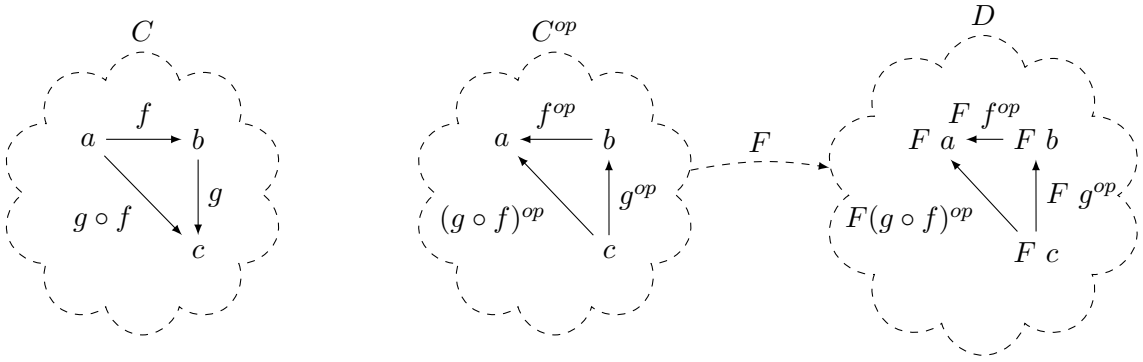
Here's one picture to visualize what's happening:



The contravariant terminology is often not used. Instead, a contravariant functor is defined as a covariant functor from the *opposite category*.

Definition 3.6. Let C be a category. The opposite category C^{op} is the category obtained by reversing all the arrows of C. Composition is defined as $f^{op} \circ g^{op} = (g \circ f)^{op}$.

With the opposite category defined, a contravariant functor $F : C \to D$ is a covariant functor $F : C^{op} \to D$.



Either terminology is fine. We can be careful to call-out contravariant functors, or we can make all contravariant functors have an opposite category as a source and just call them functors.

3.1 Bi-functors

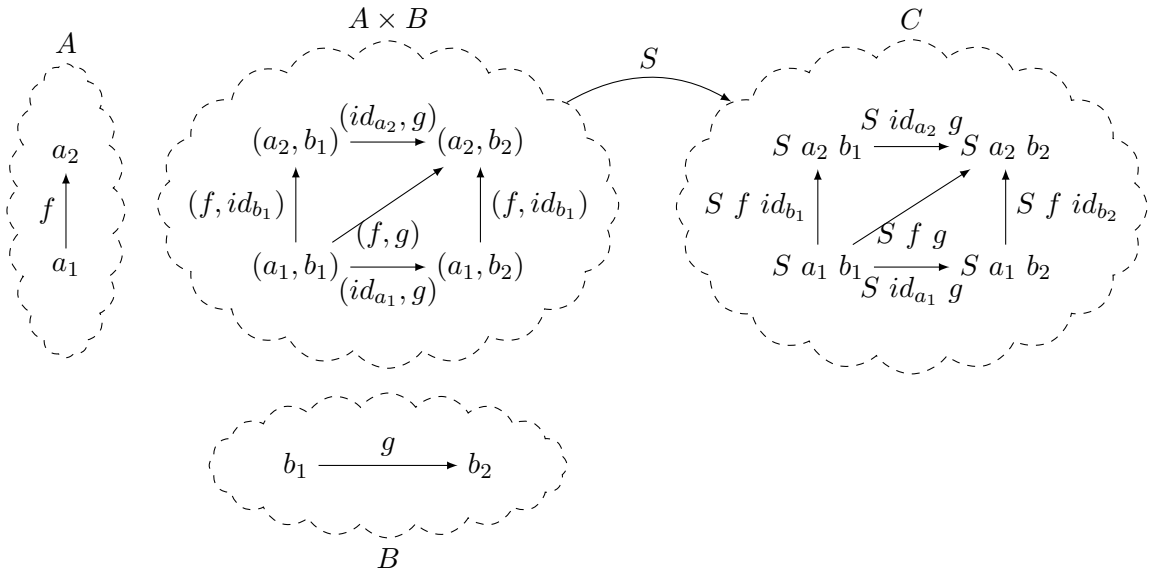
Intuitively, a bi-functor is a functor that takes two arguments, either two objects or two morphisms. A bi-functor does not necessarily have to take two objects from the same category. To formally define a bi-functor we must first define the product category.

3.1.1 The Product Category

Definition 3.7. The product category of two categories C and D is a category $C \times D$ with:

- Objects of $C \times D$ are pairs of objects (c, d), where c is an object of C and d is an object of D.
- Morphisms of $C \times D$ are pairs of morphisms $(f, g) : (c_1, d_1) \to (c_2, d_2)$ where $f : c_1 \to c_2$ is a morphism of C and $g : d_1 \to d_2$ is a morphism of D.
- Composition of morphisms is defined pairwise. $(f_2, g_2) \circ (f_1, g_1) = (f_2 \circ f_1, g_2 \circ g_1)$ for morphisms f_1 and f_2 of C and g_1 and g_2 of D.
- Identities are pairwise identities. $id_{(c,d)} = (id_c, id_d)$.

Then a bi-functor is simply a functor whose source is a product category. Consider a functor $S : A \times B \to C$. Here's an illustration of the situation.



Note that all of the diagrams in the above illustration commute. The diagram in $A \times B$ commutes due to the definition of composition in the product category, and the diagram in C commutes because functors preserve commuting diagrams. Compare the category A with C. We can see two copies of A in C. One for the object b_1 and another for b_2. That is consider a functor from A to C parametrized by an object of B. More specifically, let $L_{b_1} : A \to C$ be a functor that is the same as S but with the second argument fixed to b_1. Formally, $L_{b_1} a = S\ a\ b_1$ for all objects a of A, and $L_{b_1} f = S\ f\ id_{b_1}$ for all morphisms of A. Thus, the image of L_{b_1} is the left side of the rectangle in C and the image of a functor $L_{b_2} : A \to C$, parameterized by b_2, is the right side of the rectangle. We can define a similar class of functors $M_a : B \to C$, where M_a is the same as S but with the first argument fixed to a. M_{a_1} gives the bottom of the rectangle and M_{a_2} gives the top. Now based on these definitions we have

$$(M_{a_2} g) \circ (L_{b_1} f) = (L_{b_2} f) \circ (M_{a_1} g) = S\ f\ g$$

The above equation is just the commutative diagram in C. The left expression of the equation is taking the left-top path, and the middle expression is taking the bottom-right path.

It turns out, that the above equation is true more generally. The next theorem says that if you have a bi-functor, you can define two families of functors L and M for which the above equation

holds for all f and g. Conversely, if you have two families of functors L and M for which the above equation holds for all f and g, then you have a bi-functor.

Theorem 3.1. *Let A, B, and C be categories. For all objects a of A and b of B, let $L_b : A \to C$ and $M_a : B \to C$ be functors such that $M_a\ b = L_b\ a$ for all a and b. Then there exists a bi-functor $S : A \times B \to C$ with $S(-, b) = L_b$ and $S(a, -) = M_a$ for all a and b if and only if for every $f : a \to a'$ of A and $g : b \to b'$ of B*

$$(M_{a'} g) \circ (L_b f) = (L_{b'} f) \circ (M_a g)$$

The proof of this theorem is captured by the reasoning of the previous paragraphs and the above diagram. It might be a good exercise to make the argument more formal.

Bi-functors will be used more later, and things can get quite confusing and abstract. For example, let's say we have a bi-functor $S : A \times B \to C$. Let's say we have a good understanding of what objects and morphisms of A and B are. Unfortunately, this intuition will be broken after the application of S. For example, lets say I have morphisms $f : a \to a'$ and $g : b \to b'$, and I really understand these morphisms. What is $S\ f\ g$? Unfortunately, I don't know. It's some morphism of C, but my understanding of f and g has been lost. However, and this is piece I want you to remember, theorem 3.1 tells me that I can think of $S\ f\ g$ as separately taking f and g. That is, while $S\ f\ g$ is *not* a pair of morphisms, it still operates on a pair (f, g) and I can reason about this pair effectively. As an example, consider morphisms $f : a \to a'$ and $f' : a' \to a''$ of A and $g : b \to b'$ and $g' : b' \to b''$ of B. I may not have a good understanding of composition of C, but I do know

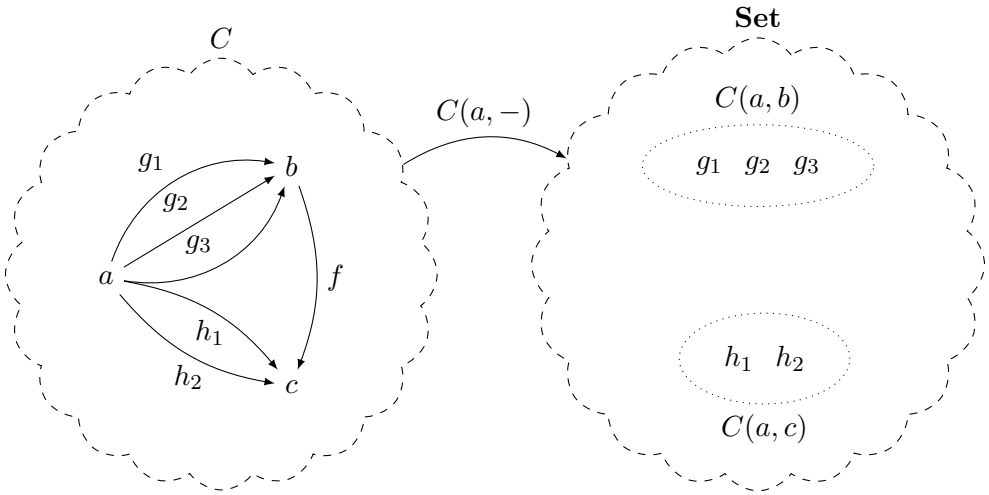
$$(S\ f'\ g') \circ (S\ f\ g) = S\ (f' \circ f, g' \circ g)$$

which uses composition of A and B which I might be more familiar with.

4 The Hom functor

I am now going to define the *Hom* functor. This section might seem a little esoteric, but the hom functor is very important in category theory. I'd rather break that ice now.

Assume we have a locally small category C. As a reminder, this means that for any two objects a and b of C the class of morphisms between a and b is a set. We use the notation $C(a, b)$ (or $Hom(a, b)$) to denote the set of morphisms between a and b. First, fix some object a of C. For any object b of C, $C(a, b)$ denotes an object of the category **Set**. Thus, for a fixed object a, $C(a, -)$ maps objects of C to objects of **Set**.

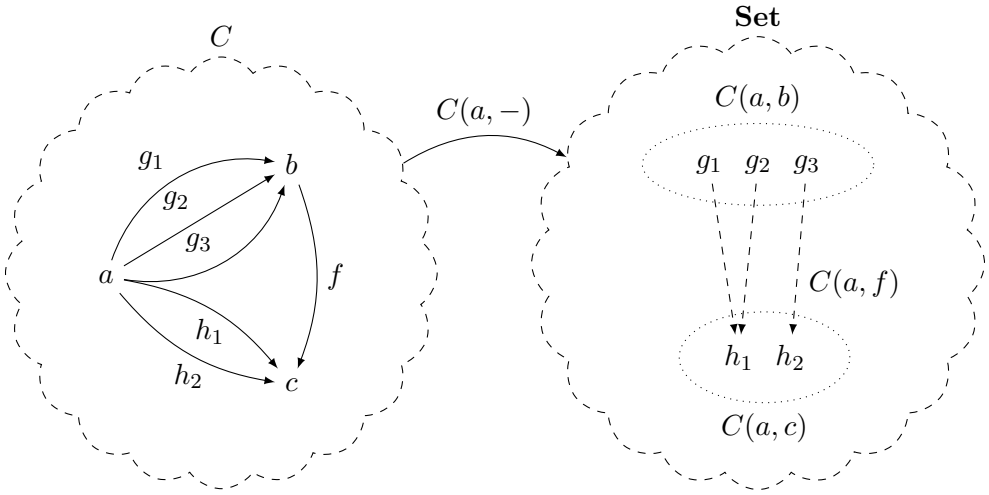


The goal is to make $C(a, -)$ a functor. Thus, it needs to map the morphism $f : b \to c$ to a function from $C(a, b)$ to $C(a, c)$. Using the above example, this resulting function needs to map an element g_1 to either h_1 or h_2. To make this choice the hom functor uses the composition of C. Consider, $f \circ g_1$. Because C is a category, this resulting morphism must be a morphism from a to c. Using the above example, let's say composition behaves as follows:

$$f \circ g_1 = h_1$$
$$f \circ g_2 = h_1$$
$$f \circ g_3 = h_2$$

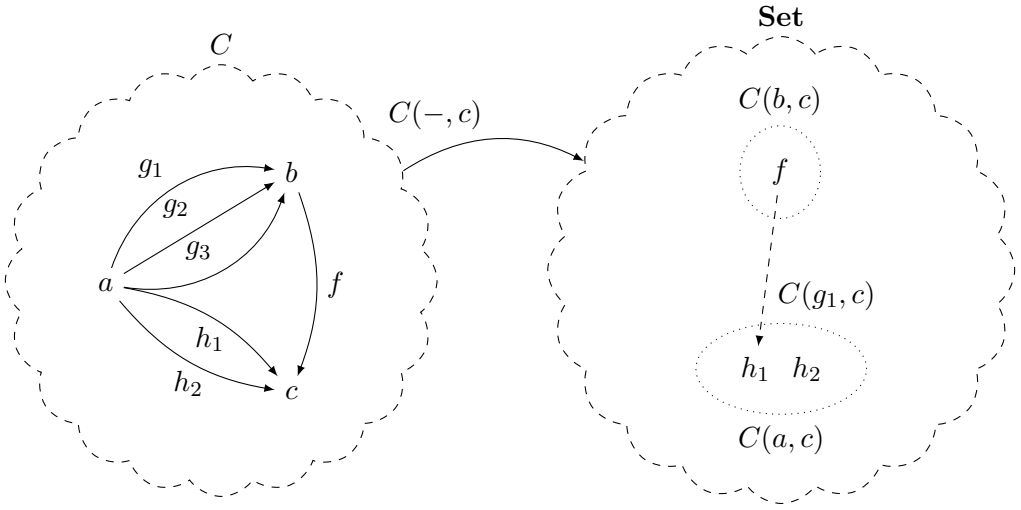
The hom functor uses this mapping to create the function from $C(a, b)$ to $C(a, c)$. That is, let $C(a, f)$ denote the mapping of f through the hom functor $C(a, -)$.

$$C(a, f)(g_1) = f \circ g_1 = h_1$$
$$C(a, f)(g_2) = f \circ g_2 = h_1$$
$$C(a, f)(g_3) = f \circ g_3 = h_2$$



In the above figure, the dashed lines in **Set** are used to represent the single function $C(a, f)$. Note that for a different morphism $f' : b \to c$, we would have a different mapping from composition and thus a different function through the hom functor. This defines the functor $C(a, -)$ for any a (this will be defined more formally in a moment).

Now consider fixing the second argument, $C(-, c)$. $C(-, c)$ maps objects of C to sets in the same way as $C(a, -)$. What about mapping morphisms? Consider $C(g_1, c)$ from above. g_1 is a morphism from a to b. If we were to follow the same direction of arrows we would create a function from $C(a, c)$ to $C(b, c)$. Given an h we would need to produce an f. There is no good way, because an f and g define an h, but not the other way around. However, we can define a function from $C(b, c)$ to $C(a, c)$ using the composition of C, by $C(g_1, c)(f) = f \circ g_1 = h_1$.



Definition 4.1. Let C be a locally-small category. The *hom functor* consists of two families of functors $C(a,-)$ and $C(-,c)$.

1. For every object a of C let $C(a,-) : C \to \mathbf{Set}$ be the following covariant functor.
   - For every object b of C, $C(a,b)$ consists of the set of morphisms $g : a \to b$.
   - For every morphism $f : b \to c$, $C(a,f) : C(a,b) \to C(a,c)$ is the function that assigns $g \rightarrowtail f \circ g$ for each g in $C(a,b)$.
2. For every object c of C let $C(-,c) : C \to \mathbf{Set}$ be the following contravariant functor.
   - For every object b of C, $C(b,c)$ consists of the set of morphisms $f : b \to c$.
   - For every morphism $g : a \to b$, $C(g,c) : C(b,c) \to C(a,c)$ is the function that assigns $f \rightarrowtail f \circ g$ for each f in $C(b,c)$.

Based on the above definition we have the following diagram commutes for all $f : a' \to a$, $g : a \to b$, and $h : b \to b'$

$$\begin{array}{ccc} C(a,b) & \xrightarrow{\quad C(f,b) \quad} & C(a',b) \\ \Big\downarrow{\scriptstyle C(a,h)} & \begin{array}{ccc} g & \rightarrowtail & g \circ f \\ \Big\downarrow & & \Big\downarrow \\ h \circ g & \rightarrowtail & h \circ g \circ f \end{array} & \Big\downarrow{\scriptstyle C(a',h)} \\ C(a,b') & \xrightarrow{\quad C(f,b') \quad} & C(a',b') \end{array}$$

The inner rectangle is meant to indicate the result of the functions of the outer rectangle on an element $g \in C(a,b)$. That is, the inner arrows don't represent morphisms. They represent functional assignments.

Because the above diagram commutes, by theorem 3.1 the hom functor is a bi-functor. Thus we might write the hom functor for category C as a bi-functor $C(-,-) : C \times C \to \mathbf{Set}$ which is contravariant in the first argument and covariant in the first. Alternatively, we can write the hom functor for category C as the bi-functor $C(-,-) : C^{op} \times C \to \mathbf{Set}$.