

7

Functors

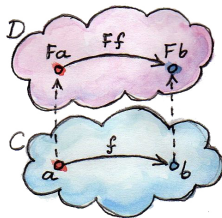
AT THE RISK OF SOUNDING like a broken record, I will say this about functors: A functor is a very simple but powerful idea. Category theory is just full of those simple but powerful ideas. A functor is a mapping between categories. Given two categories, **C** and **D**, a functor F maps objects in **C** to objects in **D** — it's a function on objects. If a is an object in **C**, we'll write its image in **D** as Fa (no parentheses). But a category is not just objects — it's objects and morphisms that connect them. A functor also maps morphisms — it's a function on morphisms. But it doesn't map morphisms willy-nilly — it preserves connections. So if a morphism f in **C** connects object a to object b ,

$$f :: a \rightarrow b$$

the image of f in **D**, Ff , will connect the image of a to the image of b :

$$Ff :: Fa \rightarrow Fb$$

(This is a mixture of mathematical and Haskell notation that hopefully makes sense by now. I won't use parentheses when applying functors to objects or morphisms.)

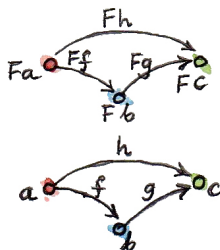


As you can see, a functor preserves the structure of a category: what's connected in one category will be connected in the other category. But there's something more to the structure of a category: there's also the composition of morphisms. If h is a composition of f and g :

$$h = g \cdot f$$

we want its image under F to be a composition of the images of f and g :


$$Fh = Fg \cdot Ff$$



Finally, we want all identity morphisms in \mathbf{C} to be mapped to identity morphisms in \mathbf{D} :

$$F \mathbf{id}_a = \mathbf{id}_{Fa}$$

Here, \mathbf{id}_a is the identity at the object a , and \mathbf{id}_{Fa} the identity at Fa .

$$F \mathbf{id}_a = \mathbf{id}_{Fa}$$


$$\mathbf{id}_a$$


Note that these conditions make functors much more restrictive than regular functions. Functors must preserve the structure of a category. If you picture a category as a collection of objects held together by a network of morphisms, a functor is not allowed to introduce any tears into this fabric. It may smash objects together, it may glue multiple morphisms into one, but it may never break things apart. This no-tearing constraint is similar to the continuity condition you might know from calculus. In this sense functors are “continuous” (although there exists an even more restrictive notion of continuity for functors). Just like functions, functors may do both collapsing and embedding. The embedding aspect is more prominent when the source category is much smaller than the target category. In the extreme, the source can be the trivial singleton category — a category with one object and one morphism (the identity). A functor from the singleton category to any other

category simply selects an object in that category. This is fully analogous to the property of morphisms from singleton sets selecting elements in target sets. The maximally collapsing functor is called the constant functor Δ_c . It maps every object in the source category to one selected object c in the target category. It also maps every morphism in the source category to the identity morphism id_c . It acts like a black hole, compacting everything into one singularity. We'll see more of this functor when we discuss limits and colimits.

7.1 Functors in Programming

Let's get down to earth and talk about programming. We have our category of types and functions. We can talk about functors that map this category into itself — such functors are called endofunctors. So what's an endofunctor in the category of types? First of all, it maps types to types. We've seen examples of such mappings, maybe without realizing that they were just that. I'm talking about definitions of types that were parameterized by other types. Let's see a few examples.

7.1.1 The Maybe Functor

The definition of **Maybe** is a mapping from type **a** to type **Maybe a**:

```
data Maybe a = Nothing | Just a
```

Here's an important subtlety: **Maybe** itself is not a type, it's a *type constructor*. You have to give it a type argument, like **Int** or **Bool**, in order to turn it into a type. **Maybe** without any argument represents a function on types. But can we turn **Maybe** into a functor? (From now on, when I speak of functors in the context of programming, I will almost

always mean endofunctors.) A functor is not only a mapping of objects (here, types) but also a mapping of morphisms (here, functions). For any function from **a** to **b**:

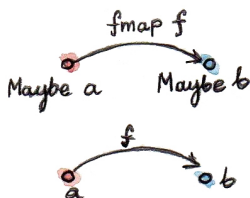
```
f :: a -> b
```

we would like to produce a function from **Maybe a** to **Maybe b**. To define such a function, we'll have two cases to consider, corresponding to the two constructors of **Maybe**. The **Nothing** case is simple: we'll just return **Nothing** back. And if the argument is **Just**, we'll apply the function **f** to its contents. So the image of **f** under **Maybe** is the function:

```
f' :: Maybe a -> Maybe b
f' Nothing = Nothing
f' (Just x) = Just (f x)
```

(By the way, in Haskell you can use apostrophes in variables names, which is very handy in cases like these.) In Haskell, we implement the morphism-mapping part of a functor as a higher order function called **fmap**. In the case of **Maybe**, it has the following signature:

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```



We often say that **fmap** *lifts* a function. The lifted function acts on **Maybe** values. As usual, because of currying, this signature may be interpreted

in two ways: as a function of one argument — which itself is a function `(a -> b)` — returning a function `(Maybe a -> Maybe b)`; or as a function of two arguments returning `Maybe b`:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Based on our previous discussion, this is how we implement `fmap` for `Maybe`:

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

To show that the type constructor `Maybe` together with the function `fmap` form a functor, we have to prove that `fmap` preserves identity and composition. These are called “the functor laws,” but they simply ensure the preservation of the structure of the category.

7.1.2 Equational Reasoning

To prove the functor laws, I will use *equational reasoning*, which is a common proof technique in Haskell. It takes advantage of the fact that Haskell functions are defined as equalities: the left hand side equals the right hand side. You can always substitute one for another, possibly renaming variables to avoid name conflicts. Think of this as either inlining a function, or the other way around, refactoring an expression into a function. Let’s take the identity function as an example:

```
id x = x
```

If you see, for instance, `id y` in some expression, you can replace it with `y` (inlining). Further, if you see `id` applied to an expression, say `id (y`

+ 2), you can replace it with the expression itself ($y + 2$). And this substitution works both ways: you can replace any expression e with `id e` (refactoring). If a function is defined by pattern matching, you can use each sub-definition independently. For instance, given the above definition of `fmap` you can replace `fmap f Nothing` with `Nothing`, or the other way around. Let's see how this works in practice. Let's start with the preservation of identity:

```
fmap id = id
```

There are two cases to consider: `Nothing` and `Just`. Here's the first case (I'm using Haskell pseudo-code to transform the left hand side to the right hand side):

```
fmap id Nothing
= { definition of fmap }
  Nothing
= { definition of id }
  id Nothing
```

Notice that in the last step I used the definition of `id` backwards. I replaced the expression `Nothing` with `id Nothing`. In practice, you carry out such proofs by “burning the candle at both ends,” until you hit the same expression in the middle — here it was `Nothing`. The second case is also easy:

```
fmap id (Just x)
= { definition of fmap }
  Just (id x)
= { definition of id }
  Just x
= { definition of id }
```

```
id (Just x)
```

Now, let's show that `fmap` preserves composition:

```
fmap (g . f) = fmap g . fmap f
```

First the `Nothing` case:

```
fmap (g . f) Nothing
= { definition of fmap }
  Nothing
= { definition of fmap }
  fmap g Nothing
= { definition of fmap }
  fmap g (fmap f Nothing)
```

And then the `Just` case:

```
fmap (g . f) (Just x)
= { definition of fmap }
  Just ((g . f) x)
= { definition of composition }
  Just (g (f x))
= { definition of fmap }
  fmap g (Just (f x))
= { definition of fmap }
  fmap g (fmap f (Just x))
= { definition of composition }
  (fmap g . fmap f) (Just x)
```

It's worth stressing that equational reasoning doesn't work for C++ style "functions" with side effects. Consider this code:


```

int square(int x) {
    return x * x;
}

int counter() {
    static int c = 0;
    return c++;
}

double y = square(counter());

```

Using equational reasoning, you would be able to inline **square** to get:

```

double y = counter() * counter();

```

This is definitely not a valid transformation, and it will not produce the same result. Despite that, the C++ compiler will try to use equational reasoning if you implement **square** as a macro, with disastrous results.

7.1.3 Optional

Functors are easily expressed in Haskell, but they can be defined in any language that supports generic programming and higher-order functions. Let's consider the C++ analog of **Maybe**, the template type **optional**. Here's a sketch of the implementation (the actual implementation is much more complex, dealing with various ways the argument may be passed, with copy semantics, and with the resource management issues characteristic of C++):

```

template<class T>
class optional {
    bool _isValid; // the tag

```

```

    T _v;
public:
    optional() : _isValid(false) {}           // Nothing
    optional(T x) : _isValid(true) , _v(x) {} // Just
    bool isValid() const { return _isValid; }
    T val() const { return _v; } };

```

This template provides one part of the definition of a functor: the mapping of types. It maps any type `T` to a new type `optional<T>`. Let's define its action on functions:

```

template<class A, class B>
std::function<optional<B>(optional<A>>>
fmap(std::function<B(A)> f) {
    return [f](optional<A> opt) {
        if (!opt.isValid())
            return optional<B>{};
        else
            return optional<B>{ f(opt.val()) };
    };
}

```

This is a higher order function, taking a function as an argument and returning a function. Here's the uncurried version of it:

```

template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}

```

There is also an option of making `fmap` a template method of `optional`. This embarrassment of choices makes abstracting the functor pattern

in C++ a problem. Should functor be an interface to inherit from (unfortunately, you can't have template virtual functions)? Should it be a curried or an uncurried free template function? Can the C++ compiler correctly infer the missing types, or should they be specified explicitly? Consider a situation where the input function `f` takes an `int` to a `bool`. How will the compiler figure out the type of `g`:

```
auto g = fmap(f);
```

especially if, in the future, there are multiple functors overloading `fmap`? (We'll see more functors soon.)

7.1.4 Typeclasses

So how does Haskell deal with abstracting the functor? It uses the type-class mechanism. A typeclass defines a family of types that support a common interface. For instance, the class of objects that support equality is defined as follows:

```
class Eq a where
    (==) :: a -> a -> Bool
```

This definition states that type `a` is of the class `Eq` if it supports the operator `(==)` that takes two arguments of type `a` and returns a `Bool`. If you want to tell Haskell that a particular type is `Eq`, you have to declare it an *instance* of this class and provide the implementation of `(==)`. For example, given the definition of a 2D `Point` (a product type of two `Floats`):

```
data Point = Pt Float Float
```

you can define the equality of points:

```
instance Eq Point where
    (Pt x y) == (Pt x' y') = x == x' && y == y'
```

Here I used the operator `(==)` (the one I'm defining) in the infix position between the two patterns `(Pt x y)` and `(Pt x' y')`. The body of the function follows the single equal sign. Once `Point` is declared an instance of `Eq`, you can directly compare points for equality. Notice that, unlike in C++ or Java, you don't have to specify the `Eq` class (or interface) when defining `Point` — you can do it later in client code. Typeclasses are also Haskell's only mechanism for overloading functions (and operators). We will need that for overloading `fmap` for different functors. There is one complication, though: a functor is not defined as a type but as a mapping of types, a type constructor. We need a typeclass that's not a family of types, as was the case with `Eq`, but a family of type constructors. Fortunately a Haskell typeclass works with type constructors as well as with types. So here's the definition of the `Functor` class:

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

It stipulates that `f` is a `Functor` if there exists a function `fmap` with the specified type signature. The lowercase `f` is a type variable, similar to type variables `a` and `b`. The compiler, however, is able to deduce that it represents a type constructor rather than a type by looking at its usage: acting on other types, as in `f a` and `f b`. Accordingly, when declaring an instance of `Functor`, you have to give it a type constructor, as is the case with `Maybe`:

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

By the way, the **Functor** class, as well as its instance definitions for a lot of simple data types, including **Maybe**, are part of the standard Prelude library.

7.1.5 Functor in C++

Can we try the same approach in C++? A type constructor corresponds to a template class, like **optional**, so by analogy, we would parameterize **fmap** with a *template template parameter* **F**. This is the syntax for it:

```
template<template<class> F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>);
```

We would like to be able to specialize this template for different functors. Unfortunately, there is a prohibition against partial specialization of template functions in C++. You can't write:

```
template<class A, class B>
optional<B> fmap<optional>(std::function<B(A)> f, optional<A> opt)
```

Instead, we have to fall back on function overloading, which brings us back to the original definition of the uncurried **fmap**:

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
  if (!opt.isValid())
    return optional<B>{};
  else
    return optional<B>{ f(opt.val()) };
}
```

```
}
```

This definition works, but only because the second argument of `fmap` selects the overload. It totally ignores the more generic definition of `fmap`.

7.1.6 The List Functor

To get some intuition as to the role of functors in programming, we need to look at more examples. Any type that is parameterized by another type is a candidate for a functor. Generic containers are parameterized by the type of the elements they store, so let's look at a very simple container, the list:

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

We have the type constructor `List`, which is a mapping from any type `a` to the type `List a`. To show that `List` is a functor we have to define the lifting of functions: Given a function `a -> b` define a function `List a -> List b`:

```
fmap :: (a -> b) -> (List a -> List b)
```

A function acting on `List a` must consider two cases corresponding to the two list constructors. The `Nil` case is trivial — just return `Nil` — there isn't much you can do with an empty list. The `Cons` case is a bit tricky, because it involves recursion. So let's step back for a moment and consider what we are trying to do. We have a list of `a`, a function `f` that turns `a` to `b`, and we want to generate a list of `b`. The obvious thing is to use `f` to turn each element of the list from `a` to `b`. How do we do this in practice, given that a (non-empty) list is defined as the `Cons` of a head and a tail? We apply `f` to the head and apply the lifted (`fmapped`) `f`

to the tail. This is a recursive definition, because we are defining `lifted f` in terms of `lifted f`:

```
fmap f (Cons x t) = Cons (f x) (fmap f t)
```

Notice that, on the right hand side, `fmap f` is applied to a list that's shorter than the list for which we are defining it — it's applied to its tail. We recurse towards shorter and shorter lists, so we are bound to eventually reach the empty list, or `Nil`. But as we've decided earlier, `fmap f` acting on `Nil` returns `Nil`, thus terminating the recursion. To get the final result, we combine the new head (`f x`) with the new tail (`fmap f t`) using the `Cons` constructor. Putting it all together, here's the instance declaration for the list functor:

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

If you are more comfortable with C++, consider the case of a `std::vector`, which could be considered the most generic C++ container. The implementation of `fmap` for `std::vector` is just a thin encapsulation of `std::transform`:

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v) {
    std::vector<B> w;
    std::transform( std::begin(v)
                    , std::end(v)
                    , std::back_inserter(w)
                    , f);
```

```
    return w;
}
```

We can use it, for instance, to square the elements of a sequence of numbers:

```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w)
           , std::end(w)
           , std::ostream_iterator(std::cout, ", "));
```

Most C++ containers are functors by virtue of implementing iterators that can be passed to `std::transform`, which is the more primitive cousin of `fmap`. Unfortunately, the simplicity of a functor is lost under the usual clutter of iterators and temporaries (see the implementation of `fmap` above). I'm happy to say that the new proposed C++ range library makes the functorial nature of ranges much more pronounced.

7.1.7 The Reader Functor

Now that you might have developed some intuitions — for instance, functors being some kind of containers — let me show you an example which at first sight looks very different. Consider a mapping of type `a` to the type of a function returning `a`. We haven't really talked about function types in depth — the full categorical treatment is coming — but we have some understanding of those as programmers. In Haskell, a function type is constructed using the arrow type constructor `(->)` which takes two types: the argument type and the result type. You've already seen it in infix form, `a -> b`, but it can equally well be used in prefix form, when parenthesized:


```
(->) a b
```

Just like with regular functions, type functions of more than one argument can be partially applied. So when we provide just one type argument to the arrow, it still expects another one. That's why:

```
(->) a
```

is a type constructor. It needs one more type **b** to produce a complete type **a -> b**. As it stands, it defines a whole family of type constructors parameterized by **a**. Let's see if this is also a family of functors. Dealing with two type parameters can get a bit confusing, so let's do some renaming. Let's call the argument type **r** and the result type **a**, in line with our previous functor definitions. So our type constructor takes any type **a** and maps it into the type **r -> a**. To show that it's a functor, we want to lift a function **a -> b** to a function that takes **r -> a** and returns **r -> b**. These are the types that are formed using the type constructor **(->) r** acting on, respectively, **a** and **b**. Here's the type signature of **fmap** applied to this case:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

We have to solve the following puzzle: given a function **f :: a -> b** and a function **g :: r -> a**, create a function **r -> b**. There is only one way we can compose the two functions, and the result is exactly what we need. So here's the implementation of our **fmap**:

```
instance Functor ((->) r) where
    fmap f g = f . g
```

It just works! If you like terse notation, this definition can be reduced further by noticing that composition can be rewritten in prefix form:

```
fmap f g = (.) f g
```

and the arguments can be omitted to yield a direct equality of two functions:

```
fmap = (.)
```

This combination of the type constructor `(->) r` with the above implementation of `fmap` is called the reader functor.

7.2 Functors as Containers

We've seen some examples of functors in programming languages that define general-purpose containers, or at least objects that contain some value of the type they are parameterized over. The reader functor seems to be an outlier, because we don't think of functions as data. But we've seen that pure functions can be memoized, and function execution can be turned into table lookup. Tables are data. Conversely, because of Haskell's laziness, a traditional container, like a list, may actually be implemented as a function. Consider, for instance, an infinite list of natural numbers, which can be compactly defined as:

```
nats :: [Integer]
nats = [1..]
```

In the first line, a pair of square brackets is Haskell's built-in type constructor for lists. In the second line, square brackets are used to create a list literal. Obviously, an infinite list like this cannot be stored in memory. The compiler implements it as a function that generates **Integers** on demand. Haskell effectively blurs the distinction between data and

code. A list could be considered a function, and a function could be considered a table that maps arguments to results. The latter can even be practical if the domain of the function is finite and not too large. It would not be practical, however, to implement `strlen` as table lookup, because there are infinitely many different strings. As programmers, we don't like infinities, but in category theory you learn to eat infinities for breakfast. Whether it's a set of all strings or a collection of all possible states of the Universe, past, present, and future — we can deal with it! So I like to think of the functor object (an object of the type generated by an endofunctor) as containing a value or values of the type over which it is parameterized, even if these values are not physically present there. One example of a functor is a C++ `std::future`, which may at some point contain a value, but it's not guaranteed it will; and if you want to access it, you may block waiting for another thread to finish execution. Another example is a Haskell `IO` object, which may contain user input, or the future versions of our Universe with "Hello World!" displayed on the monitor. According to this interpretation, a functor object is something that may contain a value or values of the type it's parameterized upon. Or it may contain a recipe for generating those values. We are not at all concerned about being able to access the values — that's totally optional, and outside of the scope of the functor. All we are interested in is to be able to manipulate those values using functions. If the values can be accessed, then we should be able to see the results of this manipulation. If they can't, then all we care about is that the manipulations compose correctly and that the manipulation with an identity function doesn't change anything. Just to show you how much we don't care about being able to access the values inside a functor object, here's a type constructor that ignores completely its argument `a`:

```
data Const c a = Const c
```

The `Const` type constructor takes two types, `c` and `a`. Just like we did with the arrow constructor, we are going to partially apply it to create a functor. The data constructor (also called `Const`) takes just one value of type `c`. It has no dependence on `a`. The type of `fmap` for this type constructor is:

```
fmap :: (a -> b) -> Const c a -> Const c b
```

Because the functor ignores its type argument, the implementation of `fmap` is free to ignore its function argument — the function has nothing to act upon:

```
instance Functor (Const c) where
    fmap _ (Const v) = Const v
```

This might be a little clearer in C++ (I never thought I would utter those words!), where there is a stronger distinction between type arguments — which are compile-time — and values, which are run-time:

```
template<class C, class A>
struct Const {
    Const(C v) : _v(v) {}
    C _v;
};
```

The C++ implementation of `fmap` also ignores the function argument and essentially re-casts the `Const` argument without changing its value:

```
template<class C, class A, class B>
Const<C, B> fmap(std::function<B(A)> f, Const<C, A> c) {
    return Const<C, B>{c._v};
}
```

Despite its weirdness, the **Const** functor plays an important role in many constructions. In category theory, it's a special case of the Δ_c functor I mentioned earlier — the endo-functor case of a black hole. We'll be seeing more of it in the future.

7.3 Functor Composition

It's not hard to convince yourself that functors between categories compose, just like functions between sets compose. A composition of two functors, when acting on objects, is just the composition of their respective object mappings; and similarly when acting on morphisms. After jumping through two functors, identity morphisms end up as identity morphisms, and compositions of morphisms finish up as compositions of morphisms. There's really nothing much to it. In particular, it's easy to compose endofunctors. Remember the function **maybeTail**? I'll rewrite it using Haskell's built in implementation of lists:

```
maybeTail :: [a] -> Maybe [a]
maybeTail [] = Nothing
maybeTail (x:xs) = Just xs
```

(The empty list constructor that we used to call **Nil** is replaced with the empty pair of square brackets **[]**. The **Cons** constructor is replaced with the infix operator **:** (colon).) The result of **maybeTail** is of a type that's a composition of two functors, **Maybe** and **[]**, acting on **a**. Each of these

functors is equipped with its own version of `fmap`, but what if we want to apply some function `f` to the contents of the composite: a `Maybe` list? We have to break through two layers of functors. We can use `fmap` to break through the outer `Maybe`. But we can't just send `f` inside `Maybe` because `f` doesn't work on lists. We have to send `(fmap f)` to operate on the inner list. For instance, let's see how we can square the elements of a `Maybe` list of integers:

```
square x = x * x

mis :: Maybe [Int]
mis = Just [1, 2, 3]

mis2 = fmap (fmap square) mis
```

The compiler, after analyzing the types, will figure out that, for the outer `fmap`, it should use the implementation from the `Maybe` instance, and for the inner one, the list functor implementation. It may not be immediately obvious that the above code may be rewritten as:

```
mis2 = (fmap . fmap) square mis
```

But remember that `fmap` may be considered a function of just one argument:

```
fmap :: (a -> b) -> (f a -> f b)
```

In our case, the second `fmap` in `(fmap . fmap)` takes as its argument:

```
square :: Int -> Int
```

and returns a function of the type:

```
[Int] -> [Int]
```

The first **fmap** then takes that function and returns a function:

```
Maybe [Int] -> Maybe [Int]
```

Finally, that function is applied to **mis**. So the composition of two functors is a functor whose **fmap** is the composition of the corresponding **fmaps**. Going back to category theory: It's pretty obvious that functor composition is associative (the mapping of objects is associative, and the mapping of morphisms is associative). And there is also a trivial identity functor in every category: it maps every object to itself, and every morphism to itself. So functors have all the same properties as morphisms in some category. But what category would that be? It would have to be a category in which objects are categories and morphisms are functors. It's a category of categories. But a category of *all* categories would have to include itself, and we would get into the same kinds of paradoxes that made the set of all sets impossible. There is, however, a category of all *small* categories called **Cat** (which is big, so it can't be a member of itself). A small category is one in which objects form a set, as opposed to something larger than a set. Mind you, in category theory, even an infinite uncountable set is considered "small." I thought I'd mention these things because I find it pretty amazing that we can recognize the same structures repeating themselves at many levels of abstraction. We'll see later that functors form categories as well.

7.4 Challenges

1. Can we turn the **Maybe** type constructor into a functor by defining:

```
fmap _ _ = Nothing
```

which ignores both of its arguments? (Hint: Check the functor laws.)

2. Prove functor laws for the reader functor. Hint: it's really simple.
3. Implement the reader functor in your second favorite language (the first being Haskell, of course).
4. Prove the functor laws for the list functor. Assume that the laws are true for the tail part of the list you're applying it to (in other words, use *induction*).