Through the Haskell Jungle

Welcome to the Haskell Jungle

In my first year of engineering school (basically last year of undergraduate school in France), I became a fixture of the computer programming club. It was a weird nerd den, but there was always something to learn there. And not too late after I started skipping classes, a friend there talked to me about Haskell. I don't remember how much I understood of the advantages and issues of pure functional programming; but I remember I liked the language, its elegance, and also its deepness of abstraction. The math kid in me liked a programming language where each new idea required a paper to express.

Flash-forward five years, and I am not a master of Haskell. I know many of the concepts at a shallow level and I toyed with it a bit, sure. Being in a formal methods research team also ensured that I learned many ideas inherent to functional programming and the like. But for Haskell itself, I never really plunged into its deepness. Never found the time.

Until today. Or a few weeks ago, to be more precise. Because as part of my reorientation to AI safety research, I now have an incentive to learn more of Haskell: MIRI wants Haskell programmers. Even if that's not exactly the job I'm aiming at, Buck Schlegeris, one of the recruiters at MIRI, confirmed in a call that it was one of the criteria MIRI used for hiring. And since I wanted to dig deeper into it for years now, that's an awesome excuse.

Which brings me to the point of this post: explaining what I'm going to do in this sequence. My goal is to learn the deep parts of Haskell, in as much detail as I need to be a good Haskell programmer. This means I want to eventually be able to:

- Write and read idiomatic Haskell code
- Optimize Haskell code
- Understand basically any Haskell code

Ideally, I would write a tutorial on every thing I study. But I want to learn fast, and writing tutorials is extremely time consuming. So I will instead post write-up of my explorations of a topic. These will generally include explanations, though without the promise of a complete narrative. Also, I will assume in the reader a general knowledge of functional programming, just not of Haskell specifics.

Finally, what will I study? Inspired by the following two study plans, I have this list for starters:

- Common Typeclasses, like Functor and Monad
- Concurrency
- Laziness
- Functional Data Structures (a la Okasaki)
- Optimization of functional programs (a la Bird)
- Principal GHC extensions
- Inner Workings of GHC

This is in no particular order, although I did begin with studying Typeclasses. Feel free to propose other topics you deem important in the comments.

With that out of the ways, let's start the trip through the Haskell jungle in the first real post, on the Functor typeclass

My Functor is Rich!

This is the first post in a sequence of write-up about my study of advanced Haskell concepts. As mentioned in the intro to this sequence, this post does not pretend to be a tutorial, merely an exploration of the subject with many explanations.

Intro

My first couple of stops on the road to Haskell proficiency are the most important typeclasses. This includes the infamous Monad, the unknown Applicative, and the subject of this post: `Functor`. But before detailing the latter, let's refresh our definition of a Haskell typeclass.

What's a typeclass?

One of Haskell's selling points is its strong type system. We might even say the focus on functional purity stems from the want for powerful type systems. Now, using strong types in a naive way creates massive redundancy: each function would have to be implemented for all the different types it works on. Of course, there will be some implementation to be done -- but the less the better.

Here, genericity comes to the rescue. We use generic types instead of concrete ones to write functions only once -- for the generic type. But this brings another set of problems: with completely generic types, we can't do anything. That's because a type tells us what we can do with its values: you can ask if an int is even or odd, but that doesn't make sense for a bool or a float. For example, a function of type

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

can only be the identity function. Why? Because I don't know anything about values of type `a`. Notably, I don't know how to build one. So when I need to return a value of type `a`, and I got one in input, my only option is to give it back.

Typeclasses are a clean way to constrain genericity, so that we can implement non trivial functions on generic types. Formally, a typeclass is defined by a set of functions:

```
class OurTypeclass a where
  f1 :: a -> [a]
  f2 :: a -> a -> Bool
```

An instance of this typeclass for a concrete type is an implementation of these functions.

```
instance OurTypeclass Bool where
  f1 x = [x]
  f2 x y = x
```

Then this instance gives us the ability to call `f1` and `f2` on values of type `Bool`. Which means we can define a generic function on any type with an instance of `OurTypeclass`, and use `f1` and `f2` in this function.

```
f :: (OurTypeclass a) => a -> a -> [a]
f x y = filter (f2 y) (f1 x)
```

The more types with instances of `OurTypeclass`, the most useful the abstraction becomes.

Another aspect of typeclasses is that instances satsify certain laws. When I say require, the laws are not actually checked at compile-time, since they tend to be undecidable in Haskell's type system. Instead, these laws are promises to Haskell programmers using the generic type. Since programmers write code assuming these promises, instances breaking the laws of their typeclass possibly break any code using this typeclass. So don't do that. Winners don't break typeclass laws.

And lastly, typeclasses generally have a minimal implementation. That is, there is a subset of functions such that all others can be defined in terms of them. So when writing an instance, you only have to implement a set of minimal functions. That being said, you can still reimplement the others; it is mostly done for optimization reasons.

That's it for the typeclass primer. What's left is the one commandment of typeclasses, stressed over and over in every resource that I scoured: **laws before burritos**. Or more seriously, **laws before intuitions**. Because it's natural to look for intuitions on what a typeclass captures. Even more when confronted with a weird one like Monad. But by limiting ourselves to these intuitions, we build our understanding on shaky grounds.

Every time I see monads (or other Haskell concepts) explained through intuition alone, I think about Terry Tao's three stages of mathematical education: pre-rigorous, rigorous, and post-rigorous. In summary, we first learn about mathematical concepts in a fuzzy and informal way (the pre-rigorous stage); then we go into the formal details, and rebuild everything from the ground up (the rigorous stage); and finally we can gloss over most details most of the time, since we can stop at any time and unwrap everything (the post-rigorous stage). Learning intuitions without looking at the functions and laws of a typeclass amounts to stagnating at the pre-rigorous stage of Haskell education. And since I aim for post-rigorous mastery, the only way is the rigorous path.

A first look at Functor

My primary source for `Functor` is the amazing Typeclassopedia, and more specifically its first section. I'll tell whenever I turn to other references.

Form and functions

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

  (<$) :: a        -> f b -> f a
  (<$) = fmap . const
```

First, one subtlety I struggled with: instances of `Functor` are functions on types. That is, a type that takes another type and gives a concrete type. A simple example is the list type `[]`: `[] a` is actually equivalent to `[a]`. Thus `[]` has an instance of `Functor`, but not `[Bool]`.

Technically, this means that the kind (the "type of a type") of a `Functor` is `* -> *`, where `*` is the kind of a concrete datatype like `Bool` or even `[a]`.

Carrying on, the meat of this class is the `fmap` function. Indeed, it constitutes a minimal definition of `Functor`: the other function `(<$)` follows by default from the definition of `fmap`. As a first step, let's look at the type of `fmap`

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

So `fmap` takes a function of type `a -> b`, a value of type `f a`, and returns a value of type `f b`. We get a transformation from one type to another, an element of our functor applied to the first type, and must transform it into an element of our functor applied to the second type. If you're able to write such a function for your type `f`, then you have a `Functor`.

The other function is just a specialization of `fmap` when the transformation is constant. So instead of giving a function, we just give an element of the return type of the function for `fmap`. (Also notice the parentheses, telling us this function use infix notation, like the + operator).

```
(<$) :: a -> f b -> f a
```

What confused me at first is that `a` and `b` are reversed compared to `fmap`. But that's only the consequence of Haskell's naming convention for types: the first type variable of kind * is named `a`, the second `b`, the third `c`... If I break this convention and use the same names than for `fmap`, I get a type fitting the description above:

```
(<$) :: b -> f a -> f b
```

As I mentioned, `(<$)` has a default implementation in terms of `fmap`.

```
(<$) = fmap . const
```

And given that the only reason to rewrite a default implementation, and that all the work is done by `fmap`, I see no reason for ever writing an explicit `(<$)`. That said, I'll be pretty interested by an counter-example if one exists.

Finally for the functions, there are utility functions defined on `Functor` instances. Theses cannot be reimplemented, as they are not part of the typeclass.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<&>) :: Functor f => f a -> (a -> b) -> f b
($>) ::  Functor f => f a -> b -> f b
void :: Functor f => f a -> f ()
```

These are not that interesting: `(<$>)` is the infix version of `fmap`, `(<&>)` and `($>)` are respectively the flipped versions of `(<$>)` and `(<$)`, and `void` just replaces the inside of a functor by the unit value `()` from the unit type.

Lawful Functor

I lied when I said that any function of type `(a -> b) -> f a -> f b` gives an acceptable `fmap`, and an acceptable instance of `Functor`. In fact, `Functor` instances must follow the laws:

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

As mentioned above, these laws are not enforced by GHC. My first instinct was that they were undecidable. After all, function equality is undecidable. But after looking a bit more into the matter, the only thing I'm sure of is that GHC can't do it. On the other hand, adding flavors of dependent types to the mix allow semi-automatic verification of these laws. Although you and I will probably never use these, or at least not in the near future.

So, these laws are not checked at compile-time. And breaking them doesn't (in general) create runtime panics and exceptions. Nonetheless, they matter: they capture what you can **expect** from a `Functor` instance. That is to say, programmers using your type as a `Functor` (through `fmap` mostly) will expect it to satisfy these laws. And their code might behave incorrectly if these assumptions don't hold.

Going back to the laws, they clarify a little more what a `Functor` is. The first one,

```
fmap id = id
```

says that when passing the identity function to `fmap`, it does nothing. This fits the intuition that a `Functor` is some sort of container, a structure with values of type `a` inside. And `fmap` can only touch these values, not the structure itself.

Let's look at the example: `map`, the instance of `fmap` for lists. It applies the function to every element of the list, but doesn't change the number of elements, or their position.

As for the second law, it constrains `Functor`s even more towards containers:

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

When applying a function `f` through `fmap` and then a function `g` through `fmap`, that must be equivalent to applying the composition `g . f` through `fmap`.

This also holds for `map`, since applying `f` then `g` to each element is the same as applying `g . f` to each element.

Examples and anti-examples

When I learn a definition, I like to get both examples and anti-examples (examples not satisfying the definition). That gives me an outline of the concept from the inside (what it requires) and from the outside (what it forbids).

Let's start with the instance for list, which I already mentioned:

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

As explained above, this instance satisfies the `Functor` laws. Can we write one that doesn't? That's pretty easy.

```
instance Functor [] where
  fmap f l = []
```

or

```
instance Functor [] where
  fmap f [] = []
  fmap f x:xs = (fmap f xs)++[f x]
```

Now, these two instances break both laws. Can we break one and not the other? We'll see later that the second law follows from the first law; so we can't break the second without breaking the first. But can we break the first without breaking the second?

If the second law is true, then

```
fmap (g . id) = (fmap g) . (fmap id)
fmap g = (fmap g) . (fmap id)
```

and

```
fmap (id . g) = (fmap id) . (fmap g)
fmap g = (fmap id) . (fmap g)
```

This entails that `fmap id` acts like the identity on elements of `f a` or `f b` depending on the side. I think this means that the second law implies the first law, but it's slightly tricky. This is because not all functions of type `f a -> f b` are representable as `fmap g` for some `g :: a -> b`. For example, the function applying `g` on each element of a list and reversing the list is not representable by an application of `fmap`. So `fmap g = (fmap g) . (fmap id)` is not equivalent to `h = h . (fmap id)`, and similarly for the other direction.

Thus I think the second law implies the first law, but I would appreciate any clean proof or counter-example.

Another `Functor` is `Maybe`. The canonical instance is

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

Here too, we can write blatantly false instances.

```
instance Functor Maybe where
  fmap f x = Nothing
```

and... that's it actually. I just realized while coming up with examples that `Maybe` only has this bad `Functor` instance. Because with `Nothing`, you can only return `Nothing` (nothing to apply `f` on), and with `Just x`, the only way to get a value of type `b` from it is by applying `f`. The other option being returning `Nothing`.

The intuition behind the `fmap` for `Maybe` is that it applies the function when there is something, and it keeps `Nothing` when there isn't. Pretty straightforward.

One last example: `Either`. The thing that disturbed me with `Either` is that its kind is actually `* -> * -> *`, because it takes the type of the `Left` (the error type) and the type of the `Right` (the result type). Which means that for getting a `Functor`, we need to provide the first parameter (the error type).

```
instance Functor (Either e) where
  fmap f (Left e) = (Left e)
  fmap f (Right x) = (Right f x)
```

The `fmap` for `Either` acts like the one for `Maybe`, except that it keeps the error if there was one before. Here, there are no false instance, because we cannot change what we do to `Right x`, and we have no way of building a value of type `e`.

You might have noticed that all the examples I gave have lawful instances of `Functor`. Is this always the case? No: one type without a `Functor` instance is the following.

```
data Type a = Type (a -> Bool)
```

Why? Because there is no way to build a value of `b -> Bool` from a value of `a -> Bool` and a value of `a -> b`. The last two types are not composable, so we're screwed.

How do we distinguish types with a `Functor` and types without? Glad you asked; let's go down the first rabbit hole.

Down the rabbit holes

Existence: covariance and contravariance

The example above, of a type without a `Functor`, I did not find myself. Instead, after failing to find such an example, I asked Google and found it on stack overflow. I convinced myself that it has no implementation for `fmap`. But what is the underlying reason? How am I supposed to know if a given type has a `Functor` instance?

Apparently, types with a `Functor` instance must be covariant according to their type parameter. And our counter-example above is contravariant, which means it has no `Functor`. So that's just two new words. What do they mean? When is a type covariant according to a type parameter, and what does it means concretely?

First, there is a typeclass with the contravariant equivalent of `fmap`

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
  (>$) :: b -> f b -> f a
  (>$) = contramap . const
```

And `Type` above has an instance of `Contravariant`:

```
instance Contravariant Type where
  contramap f (Type g) = Type (g . f)
```

The difference is that the function given to contramap goes the other way. And from a function `a -> Bool` and a function `b -> a`, composition gives a function `b -> Bool`.

The only way to have both an instance of `Functor` and of `Contravariant` is for the type parameter to be phantom, that is to say useless:

```
data Useless a = Useless

instance Functor Useless where
  fmap _ Useless = Useless
```

```
instance Contravariant Useless where
  contramap _ Useless = Useless
```

And some types have neither an instance of `Functor` nor one of `Contravariant`:

```
data Nope a = Nope (a -> a)
```

So we have four cases. Looking around, it seems the type parameter can be in one of four relations with the type itself, which entails in which case it lies:

- if `a` is covariant, then it has a `Functor` instance and no `Contravariant` instance.
- if `a` is contravariant, then it has no `Functor` instance and a `Contravariant` instance.
- if `a` is bivariant, then it has no `Functor` instance and no `Contravariant` instance.
- if `a` is invariant, then it has both a `Functor` instance and a `Contravariant` instance.

That still does not answer my question about the meaning of covariant and contravariant. For that, the best source I found, surprisingly, was the wikipedia page.

Basically, what matters is the relation between subtypes of the type parameters, and subtypes of the full type. The most interesting example is the function (at least in a purely functional context like Haskell). Let's say I have a function `f` of type `a -> b`, where `a` and `b` are fixed types. What are the subtypes of `a -> b`? That is, what are the types of objects with which I can replace `f`, and still have correct types?

Well, `f` takes a value of type `a`. This means that when I use it in my code, I will pass to it a value of this type. Now assume I have a type `A` that is a supertype of `a`. For example, `A` is a generic type and `a` is one of its concrete types. In Haskell, `A` would be the generic type associated with a typeclass for which `a` has an instance. Then I can replace `f` by a function of type `A -> b`, and the program will still compile. On the other hand, if `a` is a generic type associated with a typeclass, and, let's say `Bool` has an instance of this typeclass, I cannot replace `f` by a function of type `Bool -> b`. Because the use of `f` entails that it can receive any value of type `a`, not just values from `Bool`.

This whole reasoning is reversed for `b`: because `f` returns a value of `b`, my program will expect something of this type. Which means that if we want to change the return type, then it must be a subtype of `b`, not a supertype.

Why do we care? Well, because this gives us the definition of covariant and contravariant. A subtype of `a -> b` returns a subtype of `b`: both the type parameter and the full type vary in the same direction. We say that `a -> b` is covariant in `b`. Whereas a subtype of `a -> b` takes a supertype of `a`; this means, you guessed it, that `a -> b` is contravariant in `a`.

This gives us a way to check, given a type and its type parameter, whether the former is covariant/contravariant/bivariant/invariant in the latter. For functions, which are pretty much the only place where contravariant positions happens, the wikipedia page gives a rule of thumbs.

> a position is covariant if it is on the left side of an even number of arrows applying to it.

Easy, no? Well, it took me a discussion with a colleague expert in type theory to realize I misread this. It does not say "if it is on the left side of an even number of arrows", but "if it is on the left side of an even number of arrows **applying to it**". Why the fuss? Because this type

```
data Tricky a = Tricky (a -> (Bool -> Int))
```

has an even number of arrows on the right of `a`, but it is contravariant in `a`. This is because the last arrow doesn't apply to `a` -- it is on the other side of the principal arrow, the root of the binary tree capturing the type expression. On the other hand, the type

```
data NotTricky a = NotTricky ((a -> Bool) -> Int)
```

is covariant in `a`

```
instance Functor NotTricky where
  fmap f (NotTricky h) = (\g -> h (g . f))
```

Because from a `a -> b` and a `b -> Bool`, I can build a `a -> Bool`. And giving that to `((a -> Bool) -> Int)`, I get a `Int`. So from `f:: a -> b` and an element of `NotTricky a`, I can build an element of `NotTricky b`.

So, caveat aside, we have a rule for computing whether a type is covariant or contravariant, or both or neither, in a type parameter. That's almost what I was looking for. What is still missing is an understanding of why `fmap` requires a type that is covariant in its type parameter. I mean, I can check for the examples, like I did above. But I did not find a general explanation.

Yet this is enough for my purpose. That being said, if someone has such a reference or explanation, I'll gladly take it.

Unicity: Free Theorems!

Once settled, somewhat, the question of whether a `Functor` exists for a given type, a natural follow-up is whether there are multiple instances for this type. As a matter of fact, there is only one instance of `Functor` (satisfying the laws) per type. I read that in the Typeclassopedia, but when I followed the links... let's just say that Haskell is full of rabbit holes.

The proof of the unicity of `Functor` instances follows from the free theorem for the type of `fmap`. What is a free theorem? Well... here goes the rabbit hole. The opening in the earth is this paper by Wadler, titled "Theorems for free!". In it, he shows that given a polymorphic type, we can deduce a theorem for all values of this type. That is, just from the type, we can deduce things about any instance of it.

Remember when I said that the only function of type `a -> a` is the identity? That actually follows from the free theorem for `a -> a` : if `f :: a -> a`, then forall `g`, we have `g . f = f . g`. The only `f` satisfying this property is the identity function (with the usual caveat that only the input/output behavior matters here, so we work up to isomorphism).

So the free theorem gives a property that any function of this type must satisfy. The way we show the unicity of the `Functor` instance is by assuming the existence of one instance satisfying the functor laws. Let's `fmap` be the function defined for this instance. Then, if we have `foo :: (a -> b) -> f a -> f b` satisfying the functor laws for our type, we use the free theorem on `foo`. It is: for `f`,`g`,`h`,`k` such that `g . k = h . f`

```
fmap g . foo k = foo h . fmap f
```

Free theorems show their powers when specialized. Here, we can choose `g = h` and `f = k = id`, which maintains the condition `g . k = g = f = h . f`. Then the free theorem specializes into

```
fmap g . foo id = foo g . fmap id
```

which by the first law of functors (satisfied by both `fmap` and `foo`) gives us:

```
fmap g = foo g
```

That is, foo has exactly the same input output behavior than `fmap`.

A similar reasoning can also be used to show that `fmap` only has to satisfy the first law -- the second law follows from the first and the free theorem for `fmap`.

All of this is actually fascinating, but I don't want to get into the derivations of free theorems in this post. First because it is already long enough; and second because free theorems deserve a post of their own.

Automation: Deriving Functor Instances

The nice thing about having only one instance of `Functor`, if it exists, is that we can derive it automatically: the Derive`Functor` extension of GHC does exactly that.

Although the page I linked is quite thorough, I didn't find much to learn in it. Almost all of it seems to talk about when a `Functor` instance cannot be derived... that is when the type is contravariant or bivariant in its last type parameter. So apparently, GHC cannot derive a `Functor` instance iff there is no such instance. That's pretty cool.

Different perspectives

Now that we did our homework on the technical details, we can go through some common intuitions about `Functor`. I think we deserved it.

Is it a container? Is it a context? No, it's a Functor!

First, there are two "obvious" interpretation of a `Functor` in Haskell: as a container, and as a context.

Although most `Functor`s can be thought as either, some works really well as one or the other. For example, lists are very natural containers. A list is a sequence of values, and applying something on this container translate into applying it to its values. On the other hand, `Either` works well as a context. It captures whether everything is alright, or whether an error happened.

Most examples I have in mind fit in one of these categories. But that's not the case of all functors:

```
data SomeType a = SomeType (Bool -> a)
```

represents neither a container of values of type `a`, nor a context in which `a` is embedded. It is more of a producer of `a`. And I don't even have an intuition like that for the weirder function types where a is covariant, like `(a -> Bool) -> Int`.

Let that remind us that intuitions only go so far.

Going to the next level

Another perspective on `Functor` appears if we change parentheses slightly:

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

That is, instead of seeing `fmap` as taking a function and a functor, and returning a transformed functor, we can see it as taking a function and returning it on the functors of the types. The jargon for this is a lift. From discussions with my friends about Haskell, and category theory, and types, I know there's a lot coming on the topic of lifts. But for now, it's only a nice new perspective on `Functor`.

Category Theory is watching you

Finally, let's address the elephant in the room: the name `Functor` comes from category theory. And yet, I did not mention it during this post. Right now, I'm not sure what it teaches me. I mean, I get that functors are mapping from one category to another, and that they maintain identity and composition in the same way that the `Functor` laws do. I even get that a Haskell `Functor` is a functor on the category Hask of Haskell types (with some subtleties).

Nonetheless, I cannot see, for now, a use for this understanding. I'll keep searching, but I think that the connection brings fruits only for concepts more complex than mere functors.

On the other hand, going deeper into typeclasses made me think about the links between Haskell and category theory. I feel that the implementation of powerful typeclasses is akin to the effort in category theory for finding the right category and transformations to do the job. Then in one case you have a short generic program, and in the other you have a small generic diagram. And in both cases, when you want to understand exactly what is happening, you need to go down into the nitty gritty details hidden behind the cute abstractions: instances in Haskell and the meaning of objects and arrows in category theory.

Conclusion

There sure are a lot of things to dig in Haskell! I came in pretty sure that I knew what `Functor` was, and I still learned a lot.