Kleisli Composition

Learn about the composition of effectful functions through the work of

Bartosz Milewski

Rob Norris

Rúnar Bjarnason

Debasish Ghosh

slides by @philip_schwarz

Bartosz Milewski Introduces the **Kleisli Category**, which is based on a **Monad** and is used to model **side effects** or **non-pure functions**
He does this by showing how a Kleisli Category based on the **Writer Monad** can model the side effects of **functions that log or trace their execution**.

I want to introduce **a category that's really close to our hearts as programmers**, OK, but it is **not the category that I told you before, the category of types and functions. It is based on the category of types and functions, but it's not the same category**, and **we get to this category by solving a programming problem**. A real-life programming problem. And I am even going to use C++ instead of Haskell.

So the problem is this. We have a library of functions that we wrote, and one day the manager says there is a new requirement that I didn't tell you before, it is enforced on us by the IRS, that every function must have an audit trail, they are auditing us, so **every function has to create a little log that says 'I was called'**, maybe even with what arguments, and so on, but just to simplify things, **the name of the function, let's say, or the action that it does, has to be appended to a log. So now go and rewrite our library so that every function leaves a trail**. And **the simplest solution that comes to mind to an imperative programmer, the fast and dirty solution, is to have a global log, something that is a string**, let's say

```
    string  log = "";
```

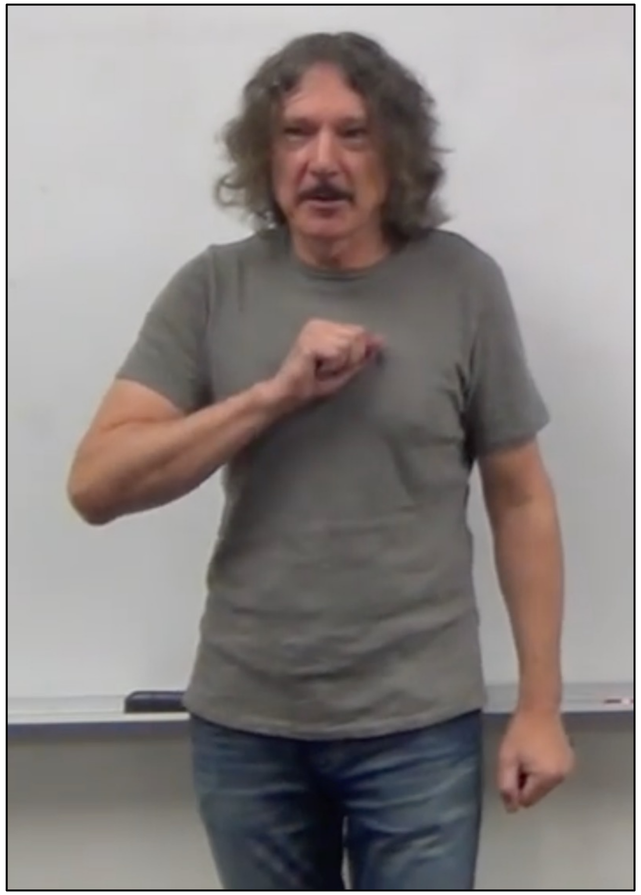
And now let's say we have a function called negate

```
    bool negate (bool x) {
      return !x;
    }
```

And now we are modifying it, just before the return, we'll insert log += "not!";

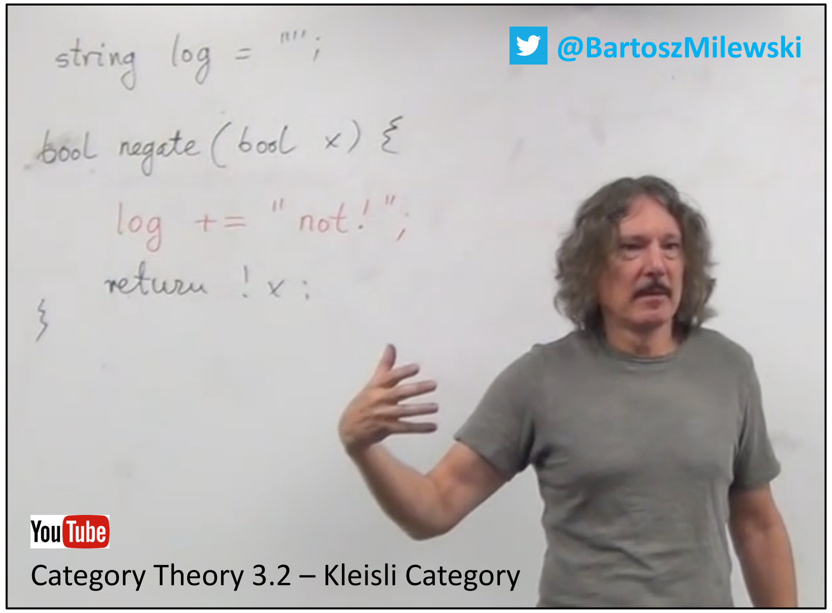
```
    bool negate (bool x) {
      log += "not!";
      return !x;
    }
```

So **the manager says, do this for every function, because this is the simplest solution**. Now what does he mean by simplest solution? We as functional programmers are thinking argh, something is wrong, but how do we explain what is wrong with this solution, because it is simple, it has the fewest lines of code, the fewest modifications, **is simplicity really measured in the number of lines of code?** If this were true why would we even bother using functions, and data structures and stuff, we would just write one huge function with a bunch of gotos, right, that's the simplest thing, right, it would probably cut down on the number of lines of code by half or so, right?



@BartoszMilewski

YouTube Category Theory 3.2 – Kleisli Category



```
string log = "";

bool negate (bool x) {
    log += "not!";
    return !x;
}
```

Category Theory 3.2 – Kleisli Category

How do we explain our argument to our manager? So the argument here is this. **If we do the above, we introduce something that is not visible in the code directly but it is something that increases the complexity of our code in a hidden way, because what it produces is a dependence.** So there is a hidden dependency between string log = “”; and log += “not!”; there is sort of a long distance interaction in quantum mechanics, right, when you look at negate, it takes a bool and returns a bool, there is no mention that it is writing to some global variable, and in fact, **if somebody removes string log = “”; then suddenly all your functions are broken even though you didn’t do anything to them, so there is this long distance dependency you have introduced**.

And if this doesn’t convince your manager then well, you do this and then **a few months later your manager comes back and says well we have another requirement, we have to use our library in a multi-threaded environment, so we’ll be calling these functions from multiple threads**.
Of course the next logical step in simplifying your life is well, let’s have a global lock, that’s the simplest solution. And of course you know, logs don’t compose and you get into possible deadlock situations, and stuff like this, **so the complexity increases even though at every step we are picking the simplest possible path, seemingly, but we have to think about this hidden complexity**.

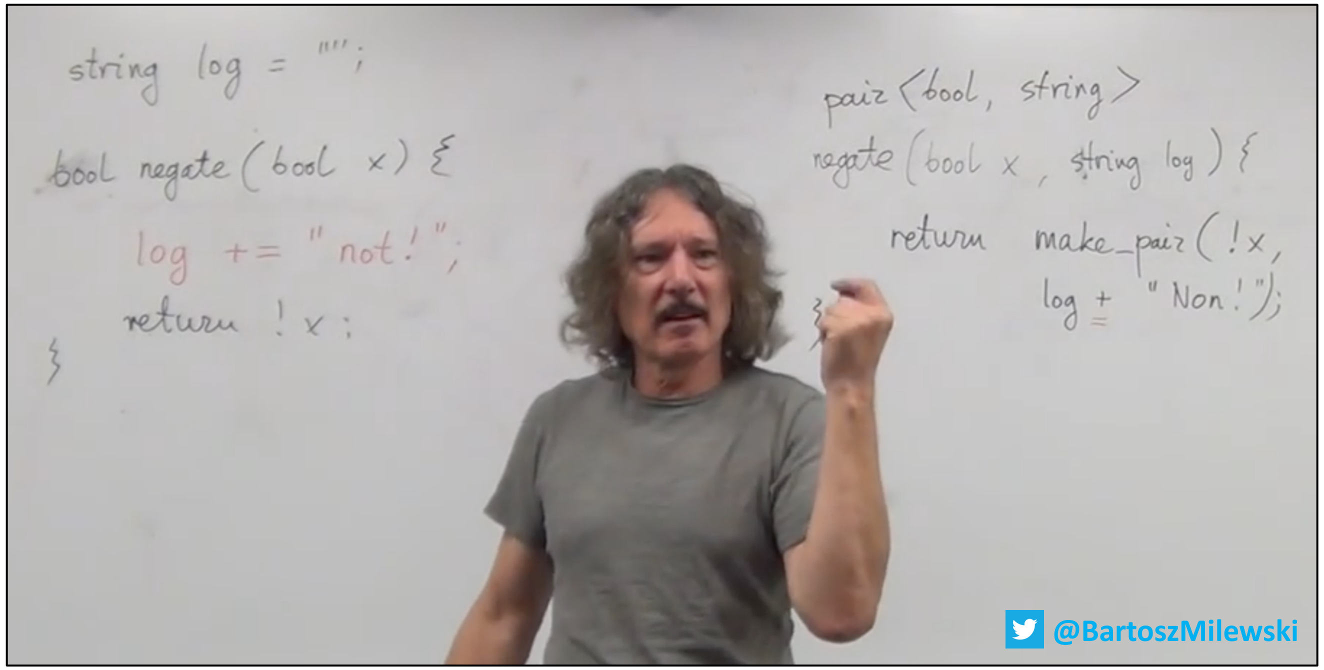
So all this stuff would not happen if we were programming in Haskell, because in Haskell you cannot have stuff like this, all functions are supposed to be pure, and can you implement this using pure functions? Of course you can. And the solution is very simple. I am going to rewrite this here. It may look more complex at first sight, but don’t be scared, because it really is simpler, it’s just that we are making things more explicit. So negate takes a bool and it takes a string log, and now it says log += “not!”; and return

```
negate (bool x, string log) {
  log += “not!”;
  return …
}
```

now what does it return, ha? Well, if it does only this then it has to return string, it has to be a pure function, well, if it is a pure function, ok, **let me write it in a purer way,** let me say return make_pair of !x and log + “not!”; and now I have to say what it returns, which is **pair<bool,string>**.

```
pair<bool, string> negate (bool x, string log) {
  return make_pair( !x, log + “not!” );
}
```

This will work, and this is a pure function, **we don’t even use +=, we use +, so I am creating a new string, everything is pure, no side effects, nothing, and this will work. It is a little bit awkward though.** Well there is one obvious awkwardness, how do you know that a function is pure? You know if you can memoize it, right? So for instance the function negate is very easy to memoize, because you just have to tabulate it: what is the value for true? What is the value for false? So it is just two elements and that’s it, you have tabulated your function, so it’s very easy to memoize. Now try to memoize this function.



```
string log = "";

bool negate (bool x) {
    log += "not!";
    return !x;
}
```

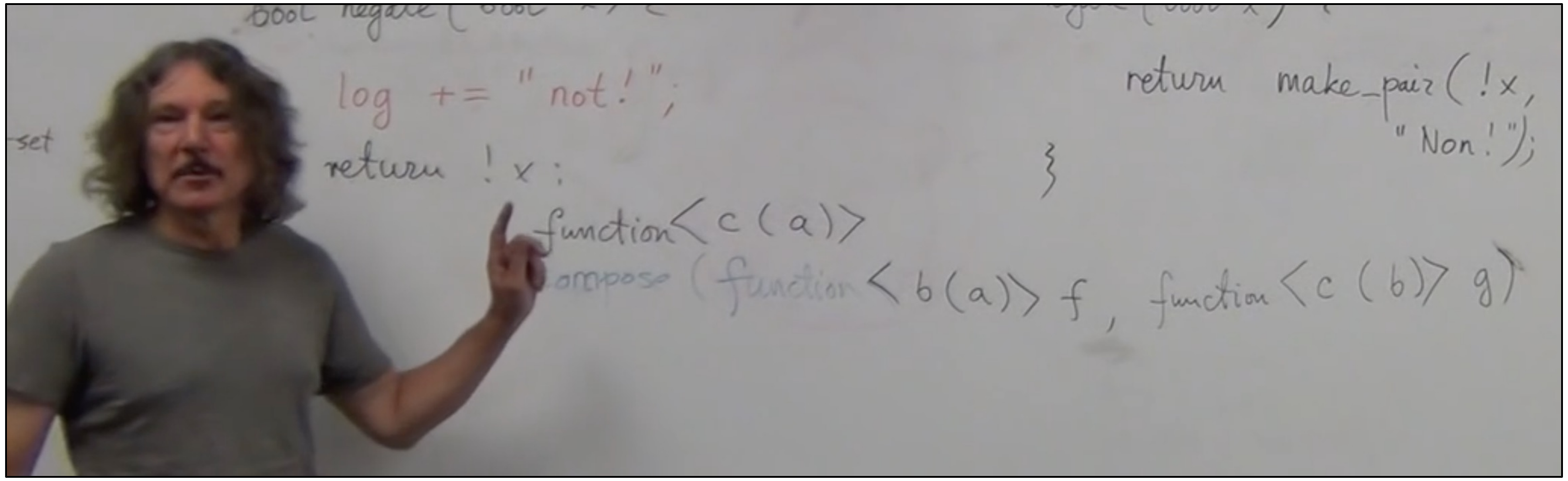
```
pair<bool, string>
negate (bool x, string log) {
    return make_pair(!x,
        log + "Non!");
}
```

YouTube Category Theory 3.2 – Kleisli Category

This is a function of two arguments, so as far as the x is concerned, there are only these two possibilities, but then every time you call this function with a different string you get a different result. So if you wanted to memoize it, you'd have to memoize it for every possible history of calls and that is not very good. **But there is also this more subtle problem with this**, it is a tiny problem, but **why does a function called negate know about appending strings**? So now every function in our library does stuff that's sort of local to it, so now we have this local and global, right, string log = ""; was a very global solution, it broke the locality, this one is more local, but still, **it has this element of knowing stuff that does not belong here**. This function should know about negation (!) and it should also know about saying "not!", so **why does it know about appending strings? That's not in the scope of what this function should know about. We are violating the principle of separation of concerns. So this is a good solution but not quite.**
What we would really like to do is for this function to just know that it has to return a pair, well it definitely has to negate this argument (x!), it also has to say "not!", but **it shouldn't take the log string as argument and do the appending**:

```
    pair<bool,string> negate (bool x) {
      return make_pair( !x, "not!" );
    }
```

Now the concerns are separated. The function does not do things that it should not be concerned with, it only does local stuff. The probem is though, who does the appending of these logs? Who generates the big log? This function does only its own thing, but somebody has to concatenate these logs. So the answer to this is, **what do we do with these functions? We compose them**, just like in every programming problem. We decompose it into functions and then we recompose the results. **So we have to be able to compose functions like these**. **So what if we modify the process of composing functions?** We have already modified the functions, but now **let's say we modify how we compose functions**.
So let's define a new way of composing functions that will take care of composing functions that return pairs.


log += "not!";
return !x;
return make_pair(!x, "Non!");
function<c(a)>
compose (function<b(a)> f, function<c(b)> g)

@BartoszMilewski

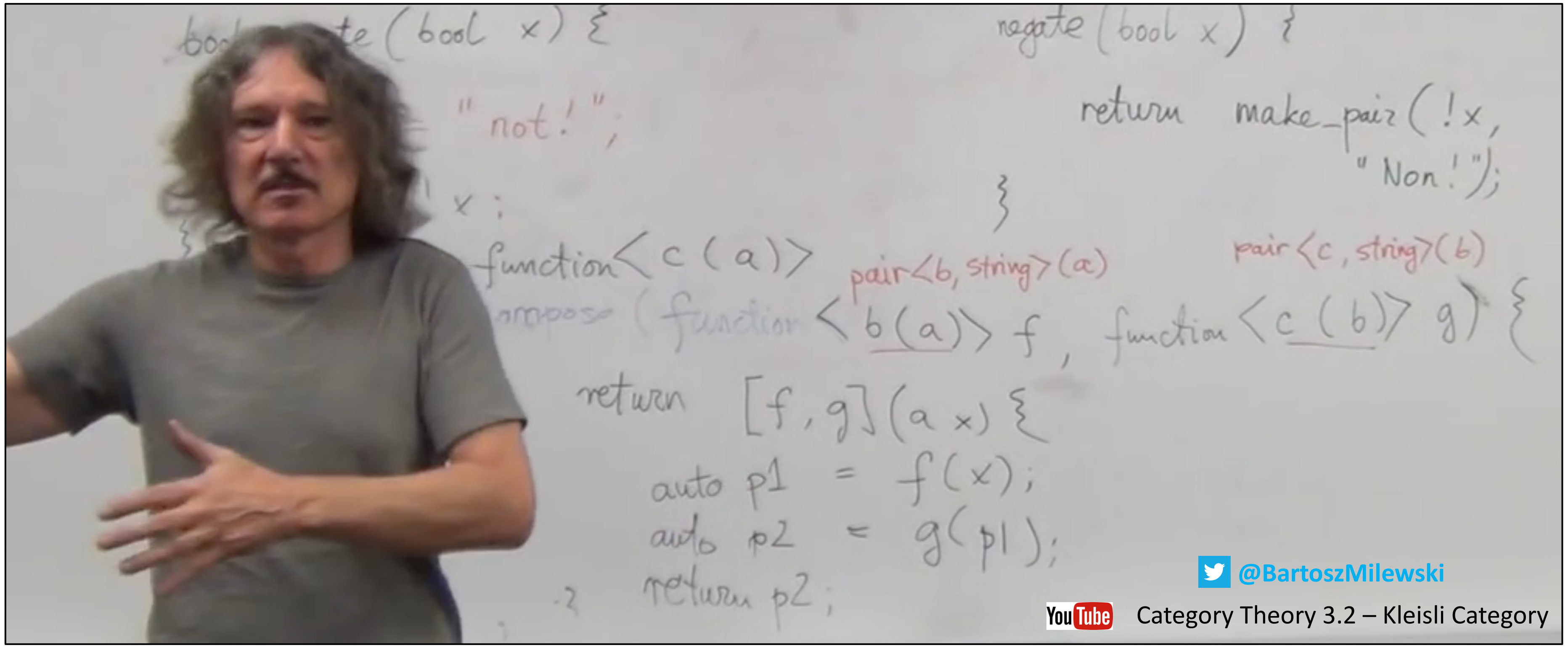
Category Theory 3.2 – Kleisli Category

So let's define a function compose, it is supposed to take two functions, so let's pass them as function objects in C++. So it has to compose functions that are composable, so it will compose a function that takes an **a** and returns a **b**, so some **f**, and it takes another function that takes a **b** and returns a **c**, let's say **g**, and it should return a third function that takes an **a**

```
    function<c(a)> compose(function<b(a)> f, function<c(b)> g)
```

So I want to define a function like this, so it's a higher order function, right, this is something that is kind of new in C++ and Java, having functions as arguments and returning functions, and creating them on the spot. **So how would we compose these two functions?** Well, we would have to call this function **f**, but where do we get the argument for it? Well, if it is returning a function, we have to, on the spot, construct a function inside compose, that's called a **lambda**, and that's a new thing in Java and also a pretty fresh thing in C++. So how do we do this? We say return, then we have to use this syntax, the capture list, what do we have to capture? We have to capture **f** and **g**, and it takes as an argument an **x** of type **a**, and here is the implementation of this function, so it is creating a function that takes an **x** of type **a**, and what does it do?
Well, it calls **f**, because it make sense, and then remembers its result, auto, let's call it **p1**. Now we have to call **g**, but what **g** is called with is the type returned by this **f**, oh, actually, so **right now I am just writing regular function composition**, so g just takes **p1** and it returns this **p2**.

```
    function<c(a)> compose(function<b(a)> f, function<c(b)> g){
      return [f, g] (a x) {
        auto p1 = f(x);
        auto p2 = g(p1);
        return p2;
      }
    }
```


```
bo    te(bool x) {
     "not!";
     x;
}
                                    negate(bool x) {
                                        return make_pair(!x,
                                                         "Non!");
                                    }
          pair<b, string>(a)              pair<c, string>(b)
function<c(a)>
compose(function<b(a)> f, function<c(b)> g) {
    return [f, g](a x) {
        auto p1 = f(x);
        auto p2 = g(p1);
        return p2;
```

That's not exactly what we want to do. This is regular function composition, that's our starting point, unfortunately they don't even define this in the standard library, like regular function composition, **but what we want to do is we want to compose functions that return pairs**, so instead of having a function **f** that returns a **b**, it returns a **pair<b, string>** and takes an **a**, and this guy **g** also returns **a pair of c and string** and takes an **a**, ok. **I am replacing these functions:**

```
function<b(a)> f
function<c(b)> g
```

with these, I'll call them embellished functions, functions that return an embellished result:

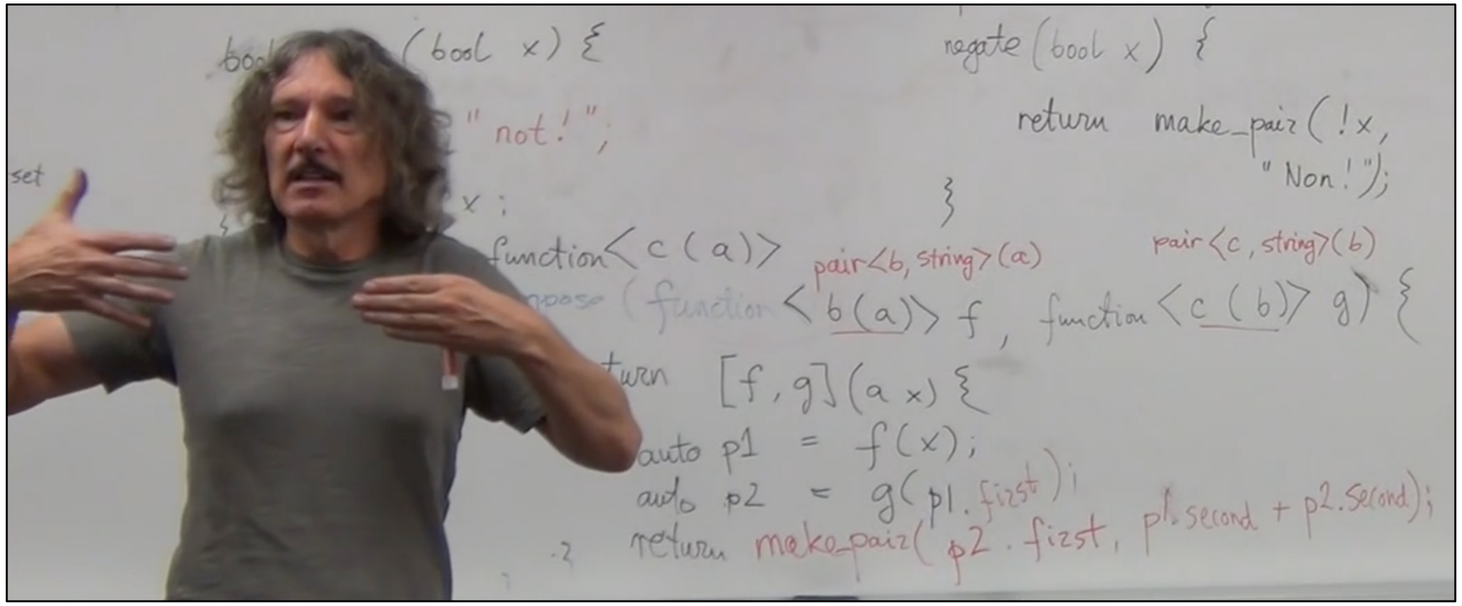
```
function<pair<b, string>(a)> f
function<pair<c, string>(b)> g
```

And the returned function needs to change to a function that takes an **a** and returns **a pair of c and string**:

```
function<pair<c, string>(a)>
```

@BartoszMilewski

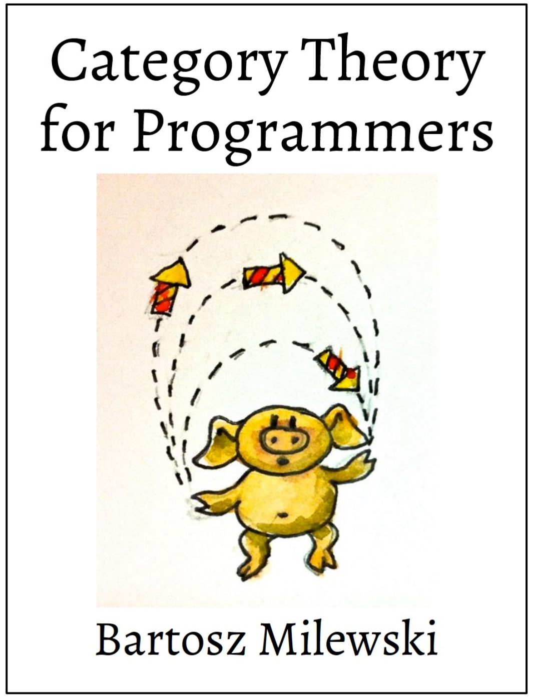
Category Theory 3.2 – Kleisli Category


```
bool (bool x) {
    "not!";
negate(bool x) {
    return make_pair(!x,
                     "Non!");
}
function<c(a)>   pair<b, string>(a)   pair<c, string>(b)
compose(function<b(a)> f, function<c(b)> g) {
    return [f, g](a x) {
        auto p1 = f(x);
        auto p2 = g(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
```

I started with functions (like negate) taking a boolean and returning a boolean, now I want a function that takes a boolean and returns a boolean paired with a string, and so on, and I want to do this for any type, so if I have a function that takes an integer and returns a double, I want to change it to a function that takes an integer and returns a double paired with a string, with part of the log. So now **p1**, if I call **f** with **x**, **p1** is a pair now, that's why I called it **p**, so if I want to call **g**, I have to **access the 'first' of this pair**, the first part of this pair is an argument to **g**. Now what I should return is **make_pair** and the final result is the first part of **p2**, so **'make_pair(p2.first'** so this was a boolean, a double, etc, I am just extracting the original value that was returned by this function, but now it also returns a string, and this is where **I am concatenating the strings**, so I take **p1.second + p2.second**:

```
function<pair<c, string>(a)>
compose(function<pair<b,string>(a)> f, function<pair<c, string>(b)> g){
  return [f, g] (a x) {
    auto p1 = f(x);
    auto p2 = g(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
  }
}
```

So what I have done here is define a new way of composing functions, and this composition take care of appending strings, so the appending is done inside the composition. It kind of makes much more sense because **appending strings really means that I am combining the results of two functions, so it really is part of composing, this is my way of composing functions.**

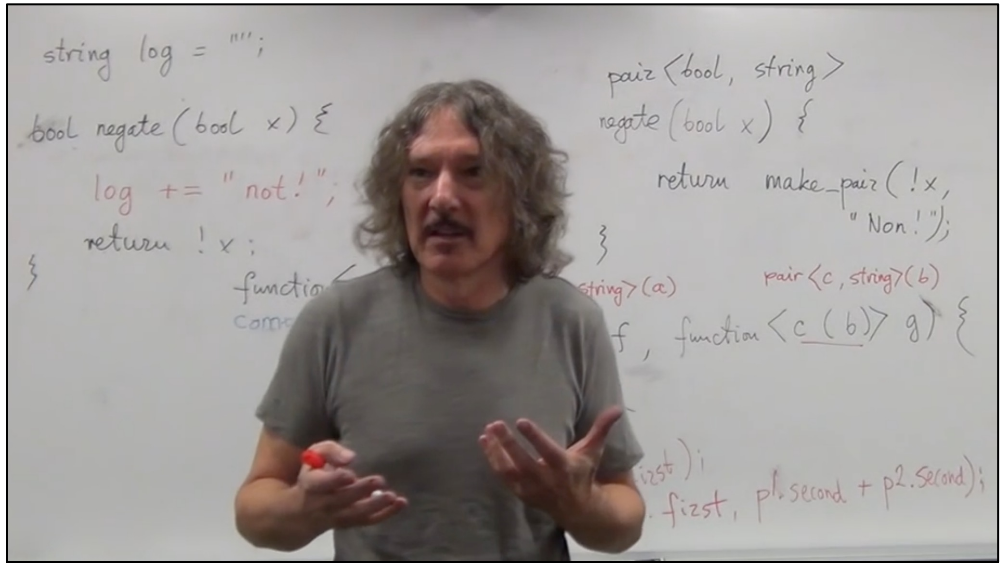

Category Theory for Programmers

Bartosz Milewski

excerpt from **Category Theory for Programmers - 4.1 The Writer Category**

So here's the **recipe for the composition of two morphisms** in this new category we are constructing:

1. Execute the **embellished function** corresponding to the first morphism
2. Extract the first component of the result pair and pass it to the **embellished function** corresponding to the second morphism
3. Concatenate the second component (the string) of the first result and the second component (the string) of the second result
4. Return a new pair combining the first component of the final result with the concatenated string.



@BartoszMilewski

Category Theory 3.2 – Kleisli Category

And now every time, **since this is a talk about category theory, if you hear the word composing, composition, you should be immediately asking, is there a category? Because a category, remember, is both composition and identity.**

I have just defined a way of composing certain things, do I have a category? So this composition of these special functions, these embellished functions, I bet it's associative, right? How do I know it is associative? Well **if you look at the first part of the pairs, I am just doing regular composition that I did before, and that one was associative**. The question is, **is the second part associative? And what is the second part? I am concatenating strings (p1.second + p2.second) and fortunately, string concatenation is associative**, so if I take three such functions and I compose them, the order in which I concatenate these logs doesn't matter. **Concatenation of logs is associative, so I have associativity because string concatenation is associative**.

Do I have identity? What would identity be for this kind of composition? It would have to be a function that returns a pair<a, string>, called id, and it takes an x of type a. And **if it is an identity it should do nothing, right? So how do you do nothing?** Well, you just return make_pair, you have to produce a pair, **our embellished functions return a pair, so our identity function has to be embellished as well**, so make a pair, and **the first part of this pair should be x, without any change, right? I am doing nothing, I am just returning whatever you pass to me, but what is the second part of this pair? The empty string, it should not append anything to the log, it should do nothing to the log.**

```
pair<a, string> id(a x) {
  return (x, “”);
}
```

And now if you remember, we have a **binary operator** (string concatenation) that is **associative** and has a **unit (“”), this thing (our new composition function and our new identity function) will work with any monoid**, it doesn't have to be string, any monoid I define is fine, so **my logging actually will work for any monoid**, it is kind of hard to abstract it in C++, but in Haskell actually the definition of this stuff includes a monoid, because we want to impose as few conditions as possible and the only condition that we have to impose is that it is a monoid, if we want composition to form a category.

So I have composition, I have identity, I have myself a category. The objects in this category are a, b, c, they are types, but arrows in this category are not my regular functions. So if I have two objects a and b, the arrow between a and b is not a function from a to b, it's a function from a to a pair, b and string (now I switched to Haskell notation).

```
a  b    a --> (b, string)
```

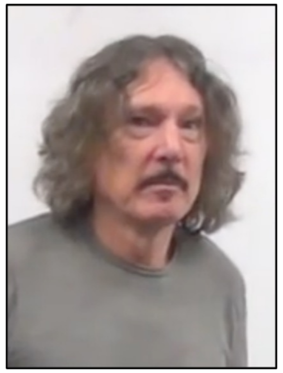
So an arrow between two objects a and b is a function that is embellished. And I know how to compose these functions, and I know what the identity is, and I have a category.

Now this category, I haven't invented this category, for this purpose.

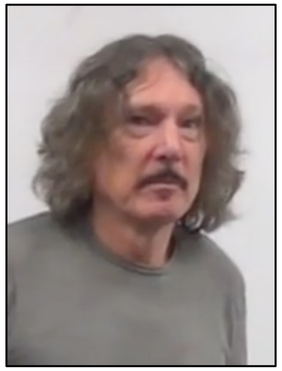
This category, is called a Kleisli category, and these are called Kleisli arrows, **these functions that are embellished**, and Kleisli arrows **can be defined for a lot of embellishments.**

I just gave you an example of one embellishment in which I am pairing my result with a string, but **there are many other possible embellishments that are extremely useful.**

This is a view of something that you have heard about before, called a monad. So these arrows actually are composable because this embellishment is a monad.



I can't really explain what a monad is yet, we'll get to this, but the important thing is that a monad is nothing special, it is just a way of composing special types of functions, **and I think that this whole problem with defining monads and people not understanding what a monad is comes from this background of imperative programming**. **In imperative programming we don't really think about composing functions**, we write our code thinking ok, I am calling this function, and it returns something, I do something with what it returns, maybe I'll call another function with it, or I'll do some operation on the spot, and so on, **whereas…**



in FP we start thinking, OK, I am composing a bunch of operations using function composition, and then you say, **what if I use a different kind of composition**, which is also function composition, but **with some kind of 'flair', something additional?** **So if I redefine composition, I have this one additional degree of freedom, If I use this additional degree of freedom, I am using a monad.**

Excerpts from '**Category Theory for Programmers**'

4. Kleisli Categories

You've seen how to model types and pure functions as a category. I also mentioned that **there is a way to model side effects, or non-pure functions, in category theory. Let's have a look at one such example: functions that log or trace their execution.** **Something that, in an imperative language, would likely be implemented by mutating some global state**…

4.1 The Writer Category

The idea of embellishing the return types of a bunch of functions in order to piggyback some additional functionality turns out to be very fruitful. We'll see many more examples of it. The starting point is our regular category of types and functions. **We'll leave the types as objects, but redefine our morphisms to be the embellished functions**…

4.2 Writer in Haskell

The same thing in Haskell is a little more terse, and we also get a lot more help from the compiler. Let's start by defining the Writer type:

type Writer a = (a, String)

Here **I'm just defining a type alias**, an equivalent of a typedef (or using) in C++. **The type Writer** is parameterized by a type variable a **and is equivalent to a pair of a and String**. The syntax for pairs is minimal: just two items in parentheses, separated by a comma. **Our morphisms are functions from an arbitrary type to some Writer type**:

a -> Writer b

We'll declare the composition as a funny infix operator, sometimes called the "fish":

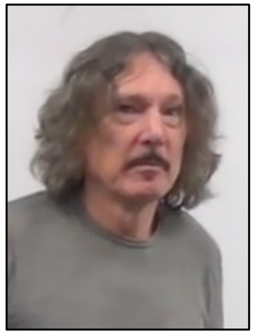
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)

It's a function of two arguments, each being a function on its own, and returning a function. The first argument is of the type (a->Writer b), the second is (b->Writer c), and the result is (a->Writer c)…

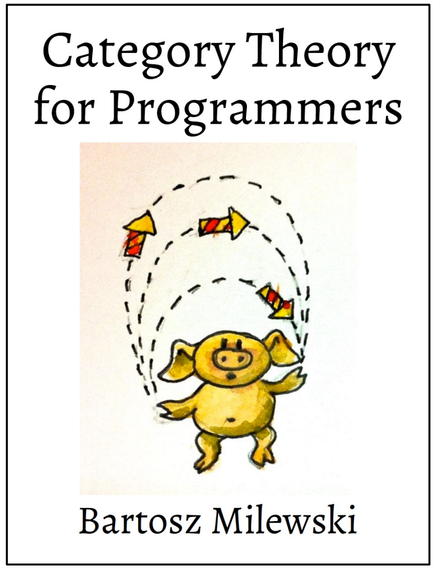
We'll declare the composition as a funny infix operator, sometimes called the "**fish**":
(**>=>**) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)

Morphisms from type A to type B are functions that go from A to a type derived from B using the particular embellishment. Each Kleisli category defines its own way of composing such morphisms

The particular monad that I used as the basis of the category in this post is called the **writer monad** and it's **used for logging or tracing the execution of functions**. **It's also an example of a more general mechanism for embedding effects in pure computations**



@BartoszMilewski



4.3 Kleisli Categories

You might have guessed that I haven't invented this category on the spot. It's an example of the so called **Kleisli category — a category based on a monad**. We are not ready to discuss monads yet, but I wanted to give you a taste of what they can do. **For our limited purposes, a Kleisli category has, as objects, the types of the underlying programming language. Morphisms from type A to type B are functions that go from A to a type derived from B using the particular embellishment. Each Kleisli category defines its own way of composing such morphisms, as well as the identity morphisms with respect to that composition.** (Later we'll see that the imprecise term "embellishment" corresponds to the notion of an endofunctor in a category.)

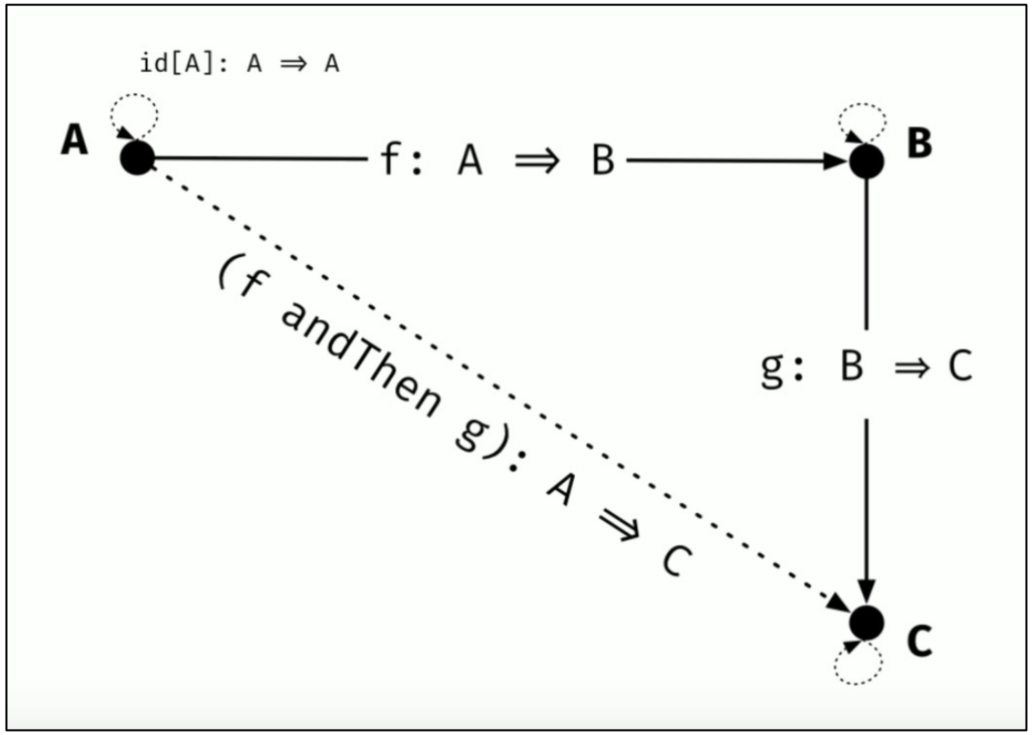
The particular monad that I used as the basis of the category in this post is called the writer monad and it's used for logging or tracing the execution of functions. It's also an example of a more general mechanism for embedding effects in pure computations. You've seen previously that we could model programming-language types and functions in the category of sets (disregarding bottoms, as usual). Here we have extended this model to a slightly different category, **a category where morphisms are represented by embellished functions, and their composition does more than just pass the output of one function to the input of another. We have one more degree of freedom to play with: the composition itself**. It turns out that this is exactly **the degree of freedom which makes it possible to give simple denotational semantics to programs that in imperative languages are traditionally implemented using side effects**.

Rob Norris @tpolecat

Rob Norris illustrates the difference between normal composition of pure functions and Kleisli composition of effectful functions with two diagrams

YouTube

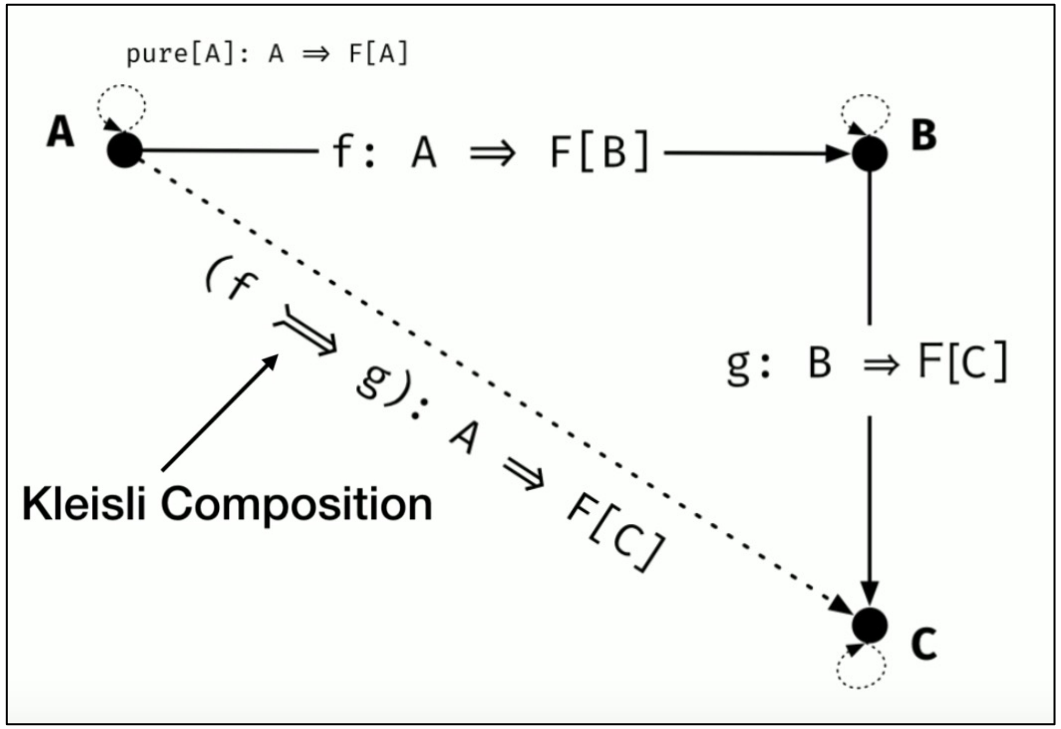
scale.bythebay.io
Rob Norris
Functional Programming with Effects



$$\begin{tikzcd}
A \arrow[loop above, dashed, "{id[A]: A \Rightarrow A}"] \arrow[rr, "{f: A \Rightarrow B}"] \arrow[rrd, dotted, "{(f\ andThen\ g): A \Rightarrow C}"'] & & B \arrow[loop above, dashed] \arrow[d, "{g: B \Rightarrow C}"] \\
& & C \arrow[loop below, dashed]
\end{tikzcd}$$

Note: **f andThen g** is the same as **g compose f**

Kleisli Category for F



$$\begin{tikzcd}
A \arrow[loop above, dashed, "{pure[A]: A \Rightarrow F[A]}"] \arrow[rr, "{f: A \Rightarrow F[B]}"] \arrow[rrd, dotted, "{(f \ggg g): A \Rightarrow F[C]}"'] & & B \arrow[loop above, dashed] \arrow[d, "{g: B \Rightarrow F[C]}"] \\
& & C \arrow[loop below, dashed]
\end{tikzcd}$$

Note: pure is another name for return (Haskell) and unit (Scala)

Bartosz Milewski introduces the need for **Kleisli composition**, in his lecture on **Monads**

A **monad** is a really simple concept.

Why do we have functions? Can't we just write one big program, with loops, if statements, expressions, assignments?
Why do we need functions? **We really need functions so that we can structure our programs**. **We need functions so that we can decompose the program into smaller pieces and recompose it**. This is what we have been talking about in category theory from the very beginning: it is **composition**.

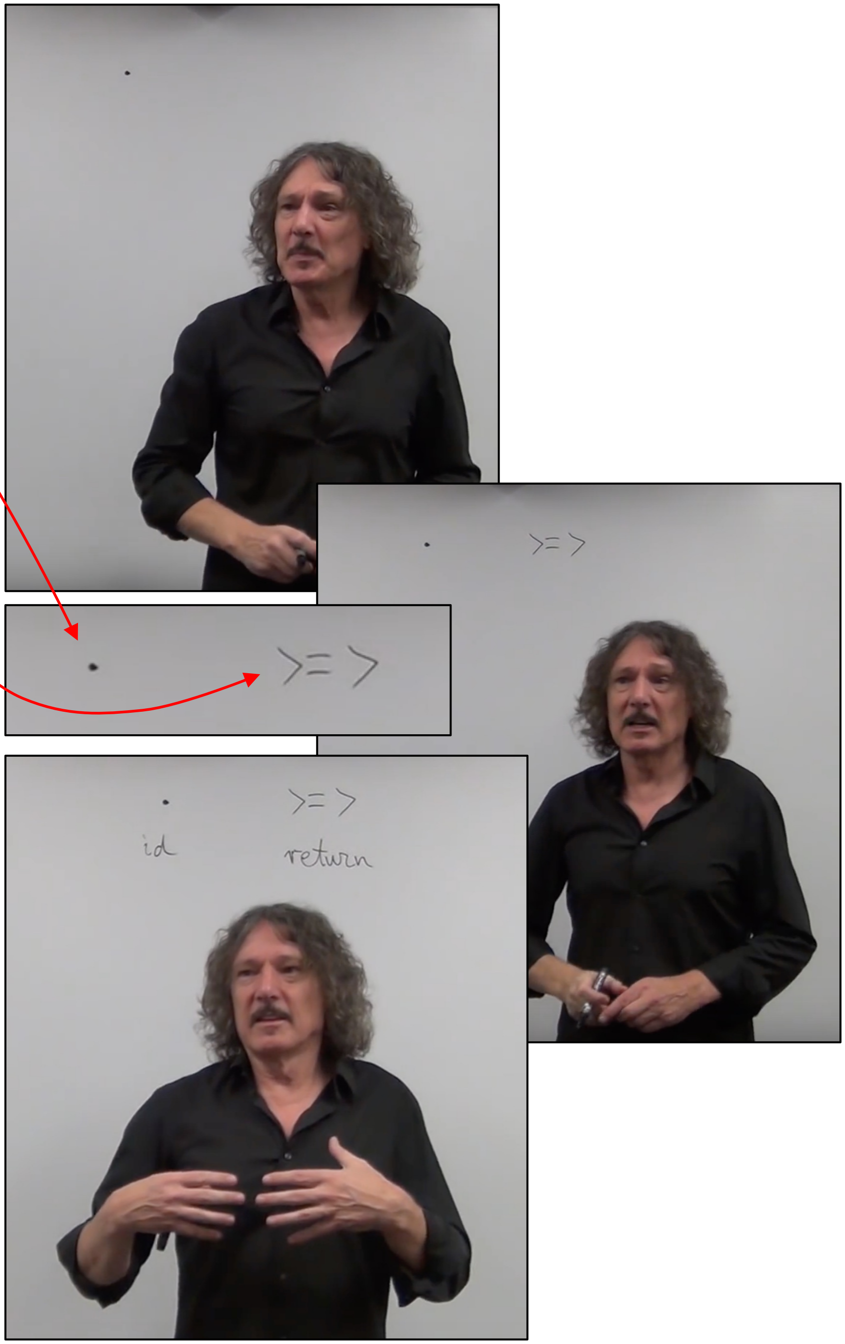
And the power of functions really is in the dot. That's where the power sits. **Dot is the composition operator** in **Haskell**.
It combines two functions so the output of one function becomes the input of the other.
So that explains what functions are really, **functions are about composition.**

And so is the monad. People start by giving examples of monads, there is the state monad, there is the exception monad, these are so completely different, what do exceptions have to do with state? What do they have to do with input/output?
Well, it's just as with functions: functions can be used to implement so many different things, **but really functions are about composition**.

And so is the monad. The monad is all about composing stuff. It replaces this dot with the Kleisli arrow…**The fish operator**.
Dot is used for composing simple functions in which the output of one function matches the input of another function, and that's the most trivial way of composing stuff.

The fish operator is used to compose these functions whose output type is embellished. So if the output of a function would be B but now **we are embellishing it with some stuff, e.g. embellishing it with logging, by adding a string to it**, **the logging kleisli arrow**, but then in order to compose these things we have to unpack the return type before we can send it on to the next function. So actually **not much is happening inside the dot**, a function is called and the result is passed to another function, but **much more is happening inside the fish, because there is the unpacking and the passing of the value to the next function, and also maybe some decision is taken, like in the case of exceptions**. Once we have this additional step of combining functions, we can make decisions, like maybe we don't want to call the next function at all, maybe we want to bypass it. So **a lot of stuff may happen inside the fish**.

And **just like we have the identity function here, that's an identity with respect to the dot, here we have this kleisli arrow that represents identity, that returns this embellished result, but of the same type, and we call it return in Haskell.** And it is called **return** because at some point you want to be able to program like an imperative programmer. So it's not that imperative programming is bad, imperative programming could be good, as long as it is well controlled, and **the monad lets you do programming that is kind of imperative style**. You don't have to do this, but sometimes it is easier to understand your code when you write it in imperative style, even though it is immediately translated into this style of composing functions. So this is just for our convenience, we want to be able to write something that looks more imperative, but behind the scene it is still **function composition upon function composition.**

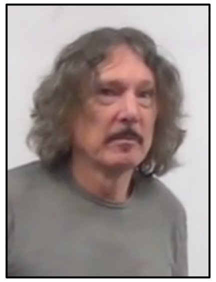


YouTube Category Theory 10.1: Monads

@BartoszMilewski

Bartosz Milewski dissects the **fish operator** (Kleisli composition) in his lecture on **Monads**

Category Theory 10.1: Monads



@BartoszMilewski

So the fish operator, the signature of the fish operator is it takes a function from a to mb. So **a** and **b** are types, **m** is this **embellishment**, and I have already mentioned that this embellishment is a functor usually, well in fact, **in any Kleisli category, this is a functor**, so the Kleisli category is still using a functor, we call this functor **m** in this case, because really, we'll see that **it is a monad, but it is a functor, it acts on a type to produce a new type**. So it is a type constructor, plus it has fmap, it knows how to lift functions.

```
(>=>) :: (a → m b)
```

So it takes one function like this and another function that goes from **b** to **m c**, and then it produces a third function that goes directly from **a** to **m c**. **So notice the mismatch. The output of the 1st function is m b but the input of the second function is b. So this fish will have to somehow, one way or another, reach into this functor m b, extract the b (, or bs, or no bs maybe), and pass it to the second function.**

```
(>=>) :: (a → m b) → (b → m c) → (a → m c)
```

So if we are implementing this guy, well let me write it in infix notation, **f >=> g**, it takes two arguments, the first function is **f**, the second function is **g**, so it takes a function **f** on the left and a function **g** on the right.

```
                  f                   g
(>=>) :: (a → m b) → (b → m c) → (a → m c)
f >=> g =
```

And what can we do? Well, first of all, we have to return a function right? (**a** --> **m c**) a function that takes an **a**. So how do you return a function from a function? Well, you have to create the function on the fly, which is using **lambda**, so you say lambda **a** arrow, so this is the notation, in haskell, for a lambda function, **a** is the argument, I am using the same letter for the argument as for the type, but types and names are from different type spaces in Haskell so it is ok, I could have used **x**, but then you wouldn't remember that it is of type **a**. So we definitely will have lambda a because we need a function that takes an **a** and return **m c**.

```
f >=> g = λa →
```

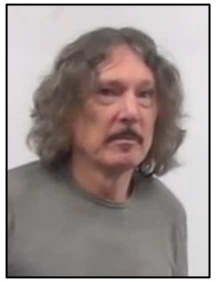
Now **once we have a and we have f, and f is from a to m b, there is really nothing else that we can do but apply this function**, we have to, there is no other way, remember, this is all polymorphic in **a**, **b** and **c**, they are completely arbitrary types, we have to be able to define the fish operator for any types **a**, **b**, **c**, so once you say that this is any type, you cannot do anything type-specific, which means you cannot do anything, except, you have a function here that takes an **a**, that's your only chance to do something with **a**, well, apply **f** to **a**, **so obviously we will apply f to a, we have no choice**.

```
f >=> g = λa →           f a
```

So the result of applying **f** to **a** is something of type **m b**. So this will produce something of type **m b** and I'll name this variable **mb** without space, because I like naming my variables using the names of types, so this is my variable **mb**. So the syntax for this is let **mb** = **f a**. It is like defining a local variable in other languages, in Haskell this is just giving a name, it is called binding, binding a name with some value.

```
f >=> g = λa → let mb = f a
```

Category Theory 10.1: Monads



@BartoszMilewski

$$(>=>) :: \overset{f}{(a \to m\,b)} \to \overset{g}{(b \to m\,c)} \to (a \to m\,c)$$

So the let expression is let something equal something in, and here we use **f a**. It makes sense to use **f a** here, otherwise we wouldn't be writing this let. So we use **f a** and **this '…' is what will be produced by the fish operator.**

$$f >=> g = \lambda a \to \text{let } mb = f\ a \text{ in } \ldots$$

So what goes in '…'? Well, we have **mb** at our disposal, and we have still untouched **g**. We have these two things, right? What we want to produce, this is a function that takes an **a** and produces **m c**. So **we need a way of taking mb and the g and produce an m c**. **So let's introduce something that does this**. This would be something that has type **m b** (with a space), a functor **m** acting on **b**, **g** is of this type **b** to **m c**, and we want to produce **m c** (**m** space **c**).

$$:: m\ b \to (b \to m\,c) \to m\ c$$

So **if we have something like this, and we'll call this function bind, and in fact let's make it an operator, a binary operator >>=.**

$$(>>=) :: m\ b \to (b \to m\,c) \to m\ c$$

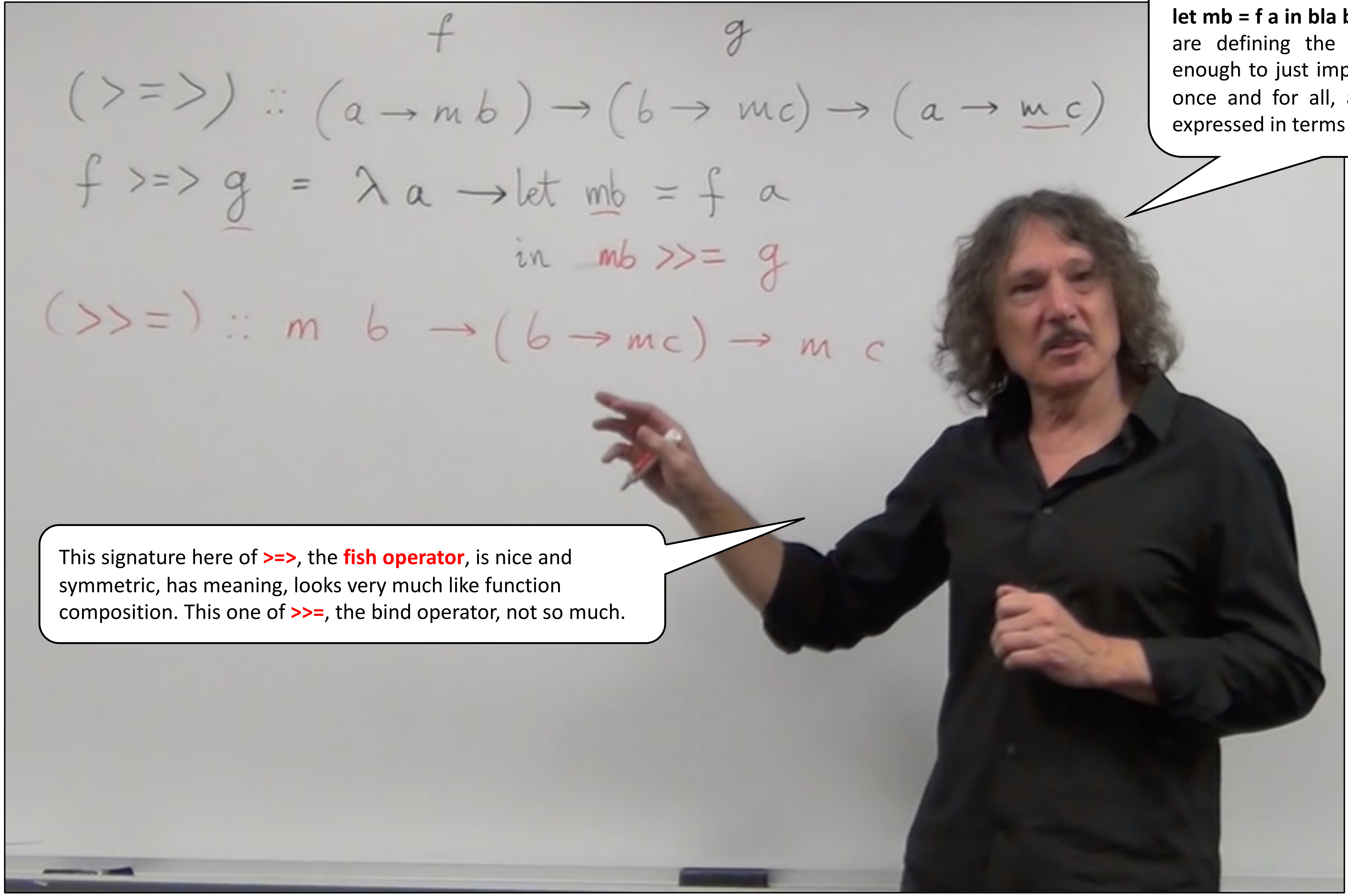
So **now in '…' we can put our mb, the binary bind operator, the g, and this will now produce our m c**. **And we are done! I mean, we have not done anything because we still have to implement this guy, the bind, right, but we have eliminated some of the boilerplate**. Instead of every time we are defining the fish operator, having to write lambda **a** --> let **mb** = **f a** in bla bla bla, **it is enough to just implement this guy, >>= once and for all, and the fish can be expressed in terms of this**.

$$f >=> \underline{g} = \lambda a \to \text{let } \underline{mb} = f\ a \text{ in } mb >>= g$$

So we have sort of simplified the problem. I don't know if this is simplification or not. **This signature here (of the fish operator), is nice and symmetric, has meaning, looks very much like function composition. This one (>>=) not so much**. This one looks like maybe a little bit like fmap, but, ok, we can live with that.

$$(>=>) :: (a \to m\,b) \to (b \to m\,c) \to (a \to m\,c)$$

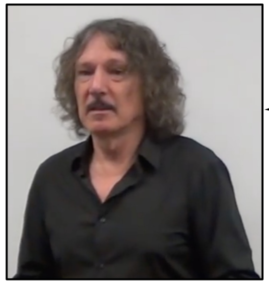
$$(>>=) :: m\ b \to (b \to m\,c) \to m\ c$$



$f \qquad g$
$(>=>) :: (a \to m\,b) \to (b \to m\,c) \to (a \to \underline{m\,c})$
$f >=> \underline{g} = \lambda a \to \text{let } \underline{mb} = f\ a$
$\text{in } mb >>= g$
$(>>=) :: m\ b \to (b \to m\,c) \to m\ c$

Instead of having to write **lambda a --> let mb = f a in bla bla bla** every time we are defining the **fish operator**, it is enough to just implement this guy, **>>=** once and for all, and the **fish** can be expressed in terms of this.

This signature here of **>=>**, the **fish operator**, is nice and symmetric, has meaning, looks very much like function composition. This one of **>>=**, the bind operator, not so much.



So a lot of people will start by saying **a monad is something that has this bind operator**, and then you ask yourself **whoever came up with this weird signature of >>= ?** And **it is not really weird, it comes from this [process we went through]**.

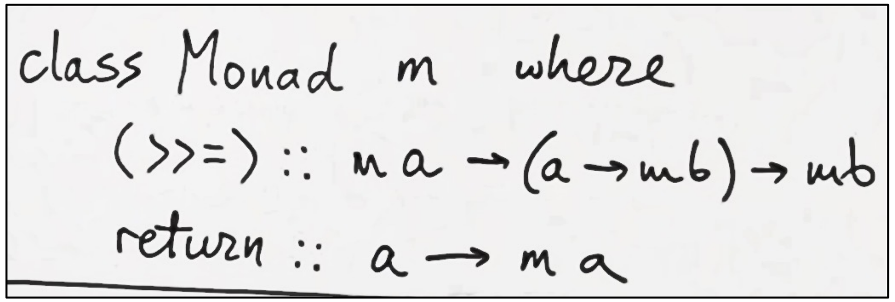
@BartoszMilewski YouTube Category Theory 10.1: Monads

So **this is how a Monad is defined in Haskell.** 'class Monad **m** where', **m** here is a type constructor, althought you don't see it here, but you will immediately see it when I define the members, which are **bind**, so we have to have bind defined, (**>>=**) :: **m a** --> (**a** --> **m b**) --> **m b**, and of course we still have **return**, which I erased, we still have this **identity Kleisli arrow** **return** **that takes an a and returns an m a, that was our kleisli arrow that served as an identity for the composition of Kleisli arrows**, but now we know that **we can recreate the composition of Kleisli arrows using bind**, any time we want to, but **we still have return to make it into a Kleisli category**.

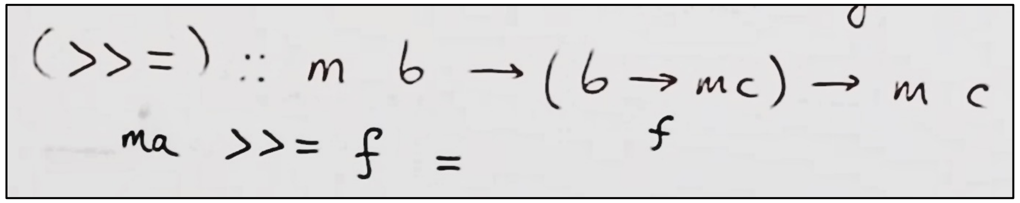
Ok, **I could stop here and say this is the definition of a Monad in Haskell**, but really this is not what most mathematicians use as a definition of a monad. **Mathematicians go deeper into the fish, they dissect the internal organs of the fish as well, and they say 'how would we implement this bind'**? Well, **remember before I said that this m is a Functor? I never used this fact that it is a functor, and in fact, if I defined monad in this way, it automatically is a functor I can define fmap using bind and return, so I don't really have to say m is a Functor, it follows from this definition**. Bind is more powerful than the Kleisli arrow. Actually, the Kleilsi arrow, **if you only assume the Kleisli arrow, you also get Functoriality**.

But suppose that we know that m is a Functor, and we want to use it, well, then how can we implement our bind? Let me again write it in infix notation. So we will have **mb** without space, ok, let me rewrite it as **ma**, I will do renaming on the fly, this is very dangerous, ok, don't do it at home, renaming variables on the fly usually ends up in you making mistakes, but since you are watching, you can correct me. So **ma** is bound (>>=) to a function, let's call it **f**, the function **f** goes from **a** to **m b**, again, renaming.

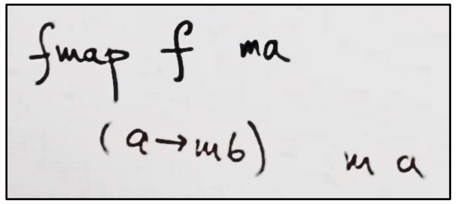
So we have these two things, and now I know that **m** is a Functor, so if it is a Functor, then I could apply this function **f** to **ma**, right, because functor lets me get inside the functor, right, I can **fmap**, aha, so I could **fmap f** over **ma**, I could, the signature of **f** is **a** to **m b**, and the signature of **ma** is **m a**.



```
class Monad m where
    (>>=) :: m a → (a → m b) → m b
    return :: a → m a
```



```
(>>=) :: m b → (b → m c) → m c
                    f
   ma >>= f =
```



```
fmap f ma
  (a→mb)  ma
```

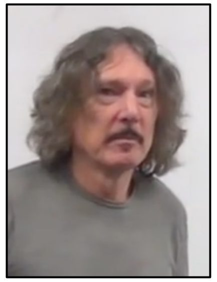
So **a** --> **m b** is my function, and with this function I can get under this **m** here [using **fmap**], and act on the **a**, which will turn this **a** into **m b**, so from this I will get fromsomething which is **m**(**m b**).

$$\underbrace{\text{fmap}\ \underset{(a \to mb)}{f}\ \underset{m\,a}{ma}}_{m(m\,b)}$$

Aha. So yes, I can do this, but I get not exactly what I wanted because I wanted to get an **m b**, not an **m**(**m b**). – I am renaming on the fly here **m b** → (**b** → **m c**) → **m c** to **m a** → (**a** → **m b**) → **m b**.

$$(>\!>=) :: m\,a \to (a \to mb) \to m\,b$$

Category Theory 10.1: Monads



And again, but if I had a function that could do this, then I could use it, so **we would need a function that does this: it turns** **m**(**m a**). I am renaming on the fly, into **m a**, right, **<u>it just strips one layer of functoriality, so it's like if you think of a Functor as a container, so we had a container with a container inside, we are saying, let me just turn it into a single container, so a container of a container becomes a single container</u>**. So this is all an operation on containment, rather than on objects sitting inside, and it is still ok, as long as I don't touch the contents, because I don't know how to touch the contents because they are polymorphic, it could be any type, I don't know how to operate on any type.

$$m(m\,a) \to m\,a$$

@BartoszMilewski

So I'll call this function **<u>join</u>**.

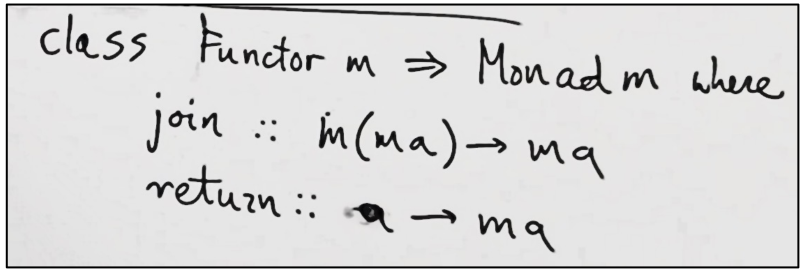
$$\text{join} :: m(m\,a) \to m\,a$$

And now you see that I can, **<u>instead of implementing bind, I can implement join</u>**, so **<u>join</u> would go here and flatten** **m**(**m b**) **to** **m b**. Oh, **<u>this is why >>= is called flatMap in some languages (e.g. Scala), because it flattens. It maps, using fmap, and then it flattens using join.</u>** Is join called flatten in Scala? Yes. It is a good name actually. So we are flattening stuff. So if we think for instance, an example of a monad is a list monad, and acting on **m a** would be a list of lists, we flatten a list of lists, we concatenate all these lists and we get a single list. So in many cases join is easy to implement. In particular, in the case of lists, join is a simple thing to implement.

$$(>\!>=) :: m\,a \to (a \to mb) \to m\,b$$

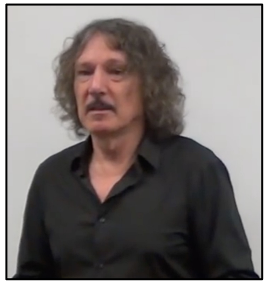
$$ma >\!>= f = \text{join}\ (\underbrace{\text{fmap}\ \underset{(a \to mb)}{f}\ \underset{m\,a}{ma}}_{m(m\,b)})$$

$$\text{join} :: m(m\,a) \to m\,a$$



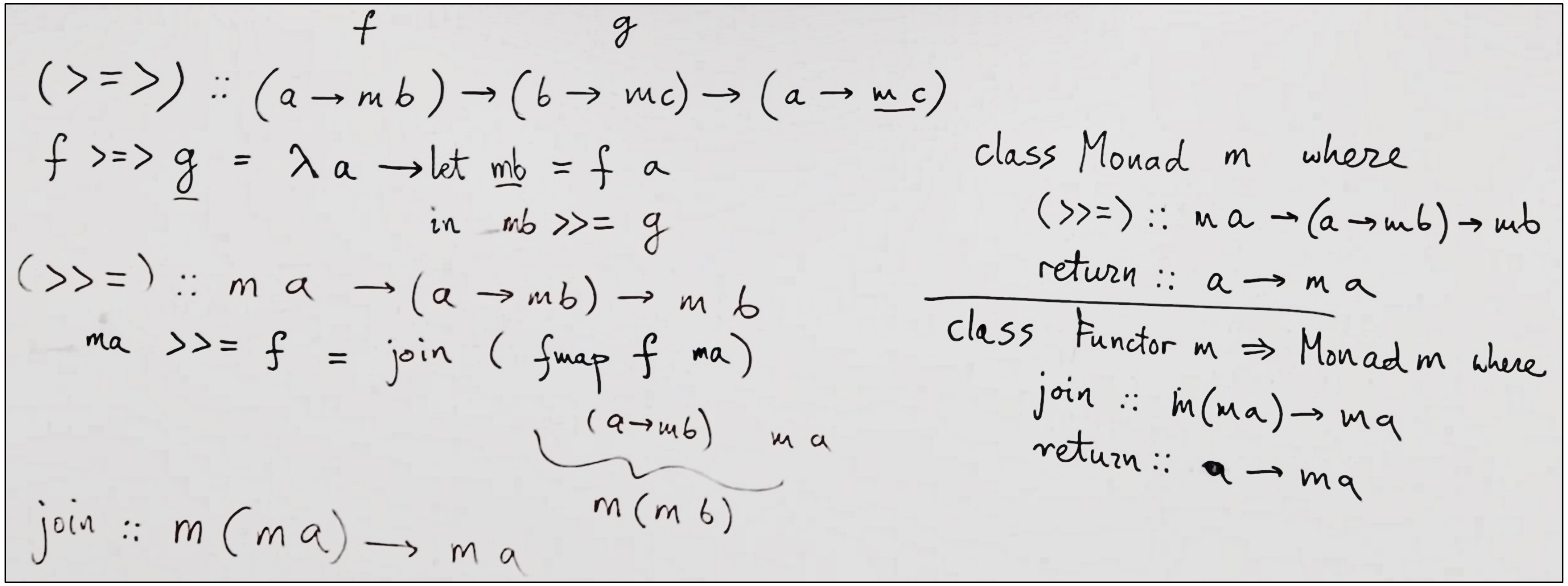
```
class Functor m => Monad m where
    join :: m(m a) -> m a
    return :: a -> m a
```

So if we wanted to define, **this is an alternative definition of a Monad**, you know, in fact in this case I have to specifically say that **m** is a **Functor**, which is actually a nice thing , that I have to explicitly specify it. Sometimes people say return is a lifting of values, right, like remember in a Functor you have a lifting of a function, this is like a lifting a values, take a value **a** and lift it to the monadic value. So in case of a list, for example, you take a value and make it a singleton list.



YouTube Category Theory 10.1: Monads

@BartoszMilewski



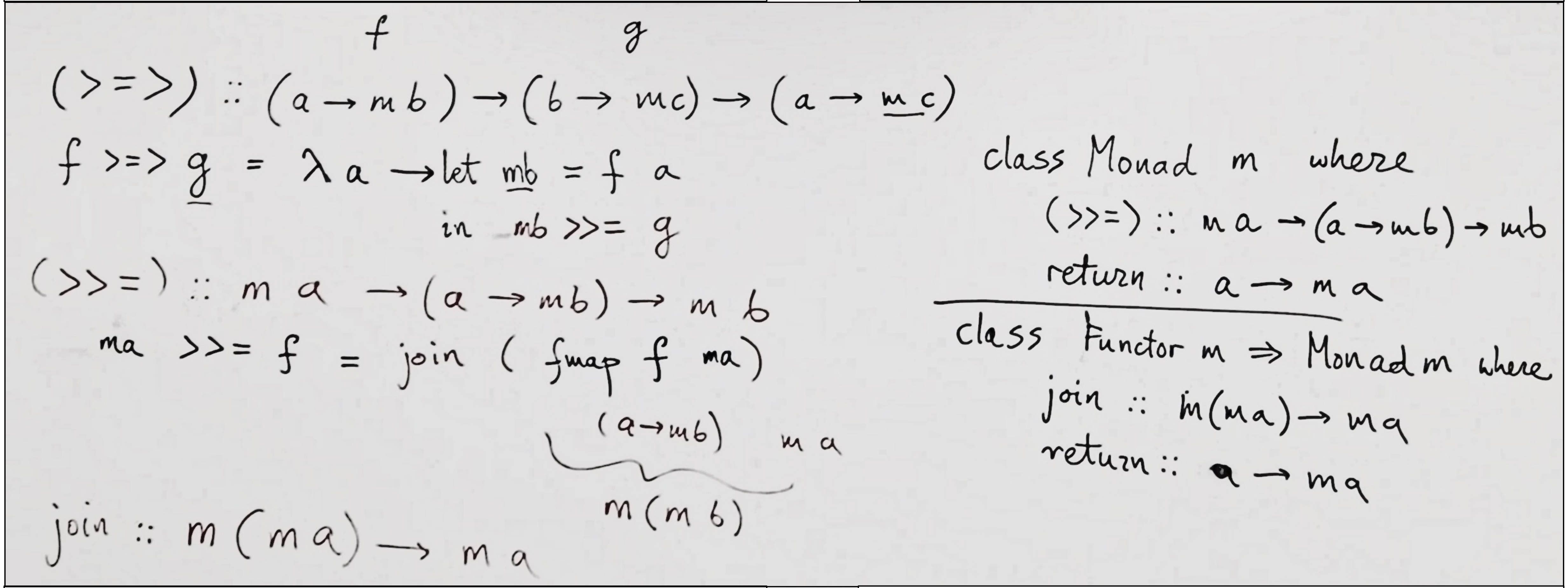
```
              f              g
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = λa -> let mb = f a
                in mb >>= g
(>>=) :: m a -> (a -> mb) -> m b
   ma >>= f = join ( fmap f ma)
                         (a->mb)  m a
                         \________________/
                              m (m b)
join :: m (m a) -> m a
```

```
class Monad m where
    (>>=) :: m a -> (a -> mb) -> mb
    return :: a -> m a
________________________________________
class Functor m => Monad m where
    join :: m(m a) -> m a
    return :: a -> m a
```

So this definition (join and return) or the definition with the Kleisli arrow, they are not used in Haskell, although they could have been.
But Haskell people decided to use this (>>= + return) as their basic definition and then for every monad they separately define join and the Kleisli arrow.
So if you have a monad you can use join and the Kleisli arrow because they are defined in the library for you.
So it's always enough to define just bind and then fish and join will be automatically defined for you.
But remember, in this case (join and return) you really have to assume that it is a functor. In this way, join is the most basic thing. Using just join and return is really more atomic than using either bind or the Kleisli arrow, because they additionally **susbsume functoriality**, whereas here, functoriality is separate, separately it is a functor and separately we define join, and separately we define return.

```haskell
class Monad m where
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  return :: a -> m a
```

```haskell
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```



$$(>=>) :: \overset{f}{(a \to m\,b)} \to \overset{g}{(b \to m\,c)} \to (a \to \underline{m}\,c)$$

$$f >=> \underline{g} = \lambda\, a \to \text{let } \underline{mb} = f\ a$$

$$\text{in } mb >>= g$$

$$(>>=) :: m\ a \to (a \to mb) \to m\ b$$

$$ma >>= f = \text{join}\ (\underbrace{\text{fmap}\ \overset{(a \to mb)}{f}\ \overset{m\,a}{ma}}_{m\,(m\,b)})$$

$$\text{join} :: m\,(m\,a) \to m\,a$$

$$\text{class Monad } m \text{ where}$$

$$(>>=) :: m\,a \to (a \to mb) \to mb$$

$$\text{return} :: a \to m\,a$$

$$\text{class Functor } m \Rightarrow \text{Monad } m \text{ where}$$

$$\text{join} :: m(ma) \to ma$$

$$\text{return} :: a \to ma$$

```
>=> Kleisli Composition (aka the fish operator)
>>= Bind

f >=> g = λa -> let mb = f a in mb >>= g
        = λa -> (f a) >>= g
```

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor m => Monad m where
  join :: m(m a) -> ma
  return :: a -> m a
```

But I haven't answered the main question.

Why? Why are we using these monads?

What problem do they solve?

Well, ok, I said **monads are used to provide composition for Kleisli arrows, right?**

But why Kleisli arrows?

Well, **we have seen one example where the Kleisli arrow was useful, when we created this logging thing, right?**

So really **the magic of the monad is not in what a monad is, because it is just composition.**

The magic is, why are these Kleisli arrows so useful?

And what kind of problem do they solve and why?

So it turns out, and that was like **a big discovery by Eugenio Moggi**, **he wrote this paper and then Philip Wadler read his paper and said yes, we can use this in Haskell, let's implement this in Haskell, and so the monad came into Haskell.**

The idea was that **in functional programming everything is pure functions, and as programmers we know that pure functions cannot express everything, things have side effects, there are so many computations in which pure functions don't really do much**, **like, basic things like input and output are not pure, getChar returns a different character every time you call it, so it is not really a function in the mathematical sense, and there are functions that access external variables, some read global state, modify global state, have side effects, then there are functions that throw exceptions, there are functions that take continuations**, **all kinds of weird things that are not pure functions and that people in imperative languages use all the time, and say we cannot program without these, if we only use pure functions then we are just heating the processor and not getting anything done.**

@BartoszMilewski

YouTube Category Theory 10.1: Monads

Notions of computation and monads
by Eugenio Moggi (1991)
https://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf

"…We do not take as a starting point for proving equivalence of programs the theory of βηconversion, which **identifies the denotation of a program (procedure) of type A → B with a total function from A to B, since this identification wipes out completely behaviours like nontermination, non-determinism or side-effects, that can be exhibited by real programs.** Instead, we proceed as follows: 1. We take category theory as a general theory of functions and develop on top **a categorical semantics of computations based on monads…**"

So it turns out, this is really a miracle, I would say, that everything that can be computed using impure functions, state, exceptions, input/output et cetera, they can all be converted into pure calculation as long as you replace regular functions with these functions that return embellished results.

So all these side effects can be translated into some kind of embellishment of the result of a function, and the function remains a pure function, but it produces more than the result, it produces a result that's hidden, embellished, encapsulated in some way, in some weird way.

So this is the interesting part: you have a computation that normally an imperative programmer would implement as an impure function and the functional programmer comes along and says I can implement this as a pure function, it's just that the output type will be a little bit different.

And it works. And this still has nothing to do with the monad.

It just says: impure computation that we do in imperative programming can be transformed into pure computations in functional programming, but they return these embellished results.

And where does the monad come in? Monads come in when we say ok, but I have these gigantic function that starts with some argument a and produces this embellished result, and do I have to just write them inline, for a 1000 lines of code, to finally produce this result?

No, I want to split it into pieces, chop it into little pieces. I want to chop a function that produces side effects into 100 functions that produce side effects and combine them, compose them.

So this is where monads come in. The monad says, well you can take this gigantic computation, pure computation, and split it into smaller pieces and then when you want to finally compose this stuff, well then use the monad.

So this is what the monad does: it just glues together, it lets you split your big computation into smaller computations and glue them together.

@BartoszMilewski

YouTube Category Theory 10.1: Monads

Excerpts from chapter 21 of '**Category Theory for Programmers**': **Monads and Effects**

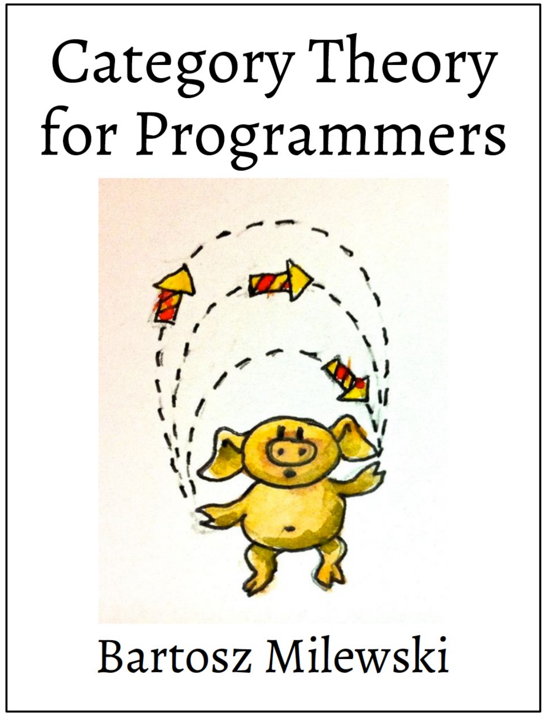
Now that we know what the monad is for — it lets us compose embellished functions — the really interesting question is why embellished functions are so important in functional programming. We've already seen one example, the Writer monad, where embellishment let us create and accumulate a log across multiple function calls. A problem that would otherwise be solved using impure functions (e.g., by accessing and modifying some global state) was solved with pure functions.

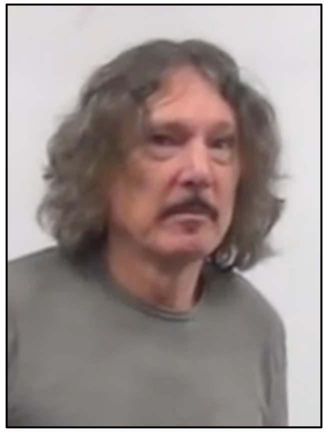
21.1 The Problem

Here is **a short list of similar problems, copied from Eugenio Moggi's seminal paper, all of which are traditionally solved by abandoning the purity of functions**:

- **Partiality**: Computations that may not terminate
- **Nondeterminism**: Computations that may return many results
- **Side effects**: Computations that access/modify state
  - – Read-only state, or the environment
  - – Write-only state, or a log
  - – Read/write state
- **Exceptions**: Partial functions that may fail
- **Continuations**: Ability to save state of the program and then restore it on demand
- **Interactive Input**
- **Interactive Output**

What really is mind blowing is that all these problems may be solved using the same clever trick: turning to embellished functions. Of course, the embellishment will be totally different in each case. You have to realize that, **at this stage, there is no requirement that the embellishment be monadic. It's only when we insist on composition — being able to decompose a single embellished function into smaller embellished functions — that we need a monad**. Again, **since each of the embellishments is different, monadic composition will be implemented differently, but the overall pattern is the same. It's a very simple pattern: composition that is associative and equipped with identity.**

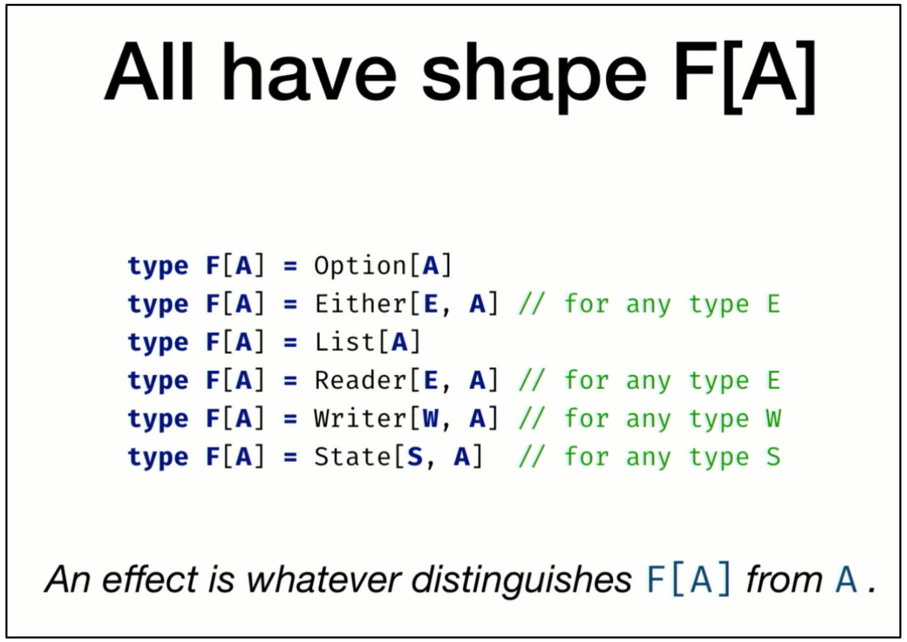




@BartoszMilewski

Six Effects

- Partiality?
- Exceptions?
- Nondeterminism?
- Dependency injection?
- Logging?
- Mutable state?



All have shape F[A]

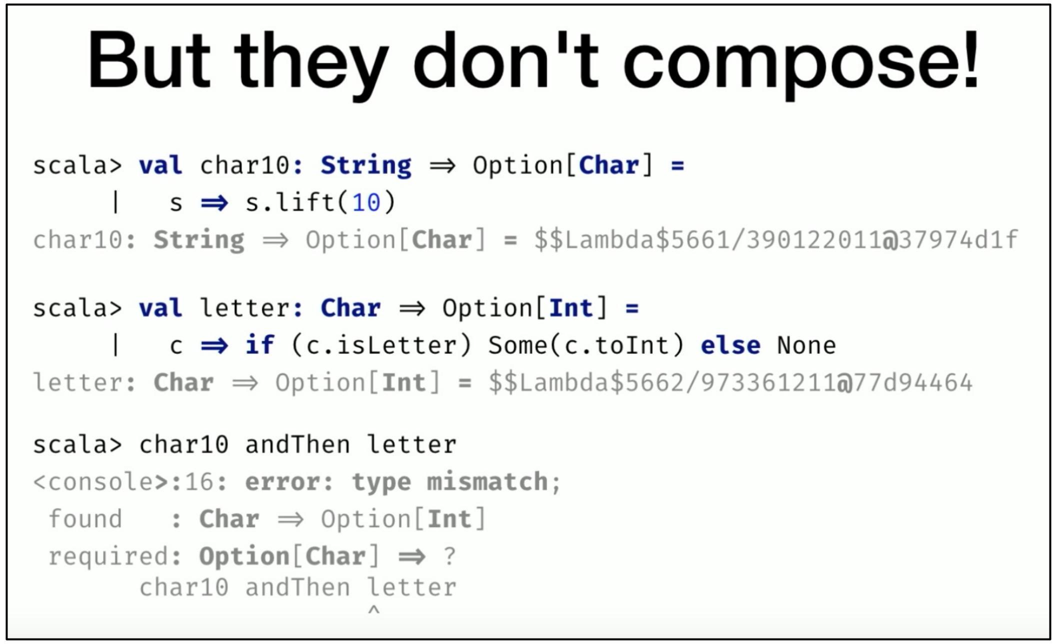
```
type F[A] = Option[A]
type F[A] = Either[E, A] // for any type E
type F[A] = List[A]
type F[A] = Reader[E, A] // for any type E
type F[A] = Writer[W, A] // for any type W
type F[A] = State[S, A]  // for any type S
```

An effect is whatever distinguishes `F[A]` *from* `A` *.*

Rob Norris @tpolecat

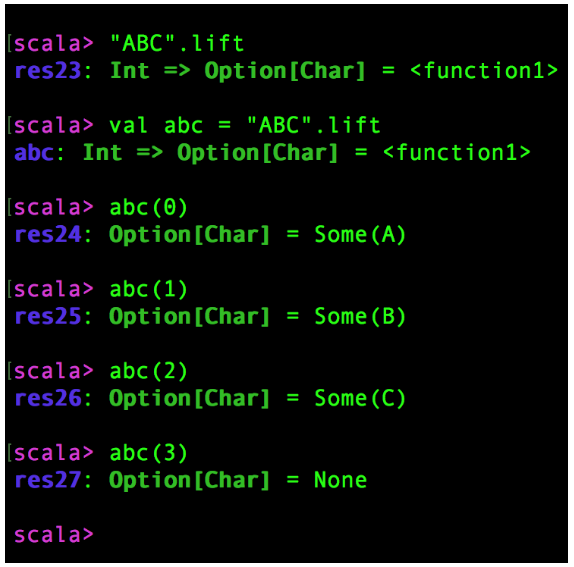
Rob Norris explaining that effectful functions, e.g. two functions both returning an **Option**, cannot be composed using normal composition, there is a type mismatch between the output of the first function and the input of the second function.

scale.bythebay.io
Rob Norris
Functional Programming with Effects

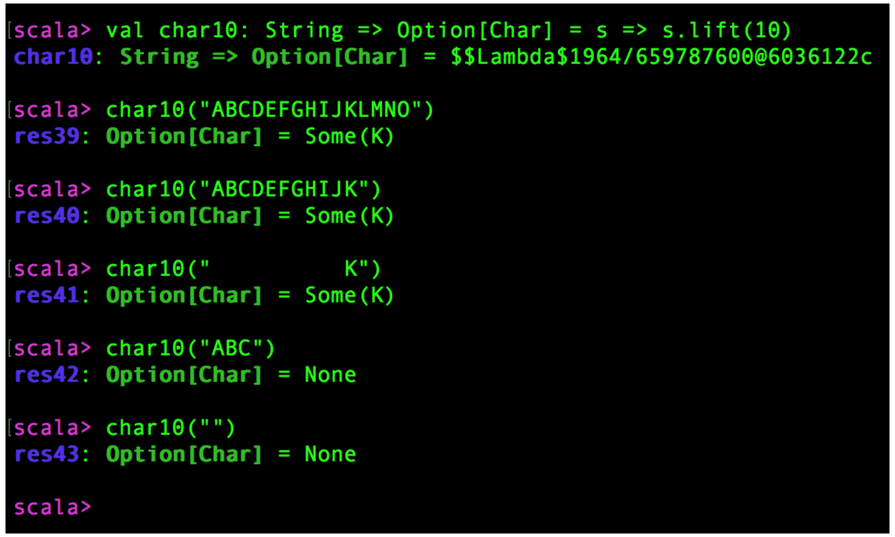


But they don't compose!

```
scala> val char10: String ⇒ Option[Char] =
     |   s ⇒ s.lift(10)
char10: String ⇒ Option[Char] = $$Lambda$5661/390122011@37974d1f

scala> val letter: Char ⇒ Option[Int] =
     |   c ⇒ if (c.isLetter) Some(c.toInt) else None
letter: Char ⇒ Option[Int] = $$Lambda$5662/973361211@77d94464

scala> char10 andThen letter
<console>:16: error: type mismatch;
 found   : Char ⇒ Option[Int]
 required: Option[Char] ⇒ ?
       char10 andThen letter
                      ^
```



```
scala> "ABC".lift
res23: Int => Option[Char] = <function1>

scala> val abc = "ABC".lift
abc: Int => Option[Char] = <function1>

scala> abc(0)
res24: Option[Char] = Some(A)

scala> abc(1)
res25: Option[Char] = Some(B)

scala> abc(2)
res26: Option[Char] = Some(C)

scala> abc(3)
res27: Option[Char] = None

scala>
```



```
scala> val char10: String => Option[Char] = s => s.lift(10)
char10: String => Option[Char] = $$Lambda$1964/659787600@6036122c

scala> char10("ABCDEFGHIJKLMNO")
res39: Option[Char] = Some(K)

scala> char10("ABCDEFGHIJK")
res40: Option[Char] = Some(K)

scala> char10("          K")
res41: Option[Char] = Some(K)

scala> char10("ABC")
res42: Option[Char] = None

scala> char10("")
res43: Option[Char] = None

scala>
```

```scala
// A typeclass that describes type constructors that allow composition with ⟹
trait Fishy[F[_]] {

  // Our identity, A ⇒ F[A] for any type A
  def pure[A](a: A): F[A]

  // Composition - the "fish" operator
  def ⟹[A, B, C](f: A ⇒ F[B], g: B ⇒ F[C]): A ⇒ F[C]

}
```

```scala
// A typeclass that describes type constructors that allow composition with ⟹
trait Fishy[F[_]] {

  // Our identity, A ⇒ F[A] for any type A
  def pure[A](a: A): F[A]

  // Composition - the "fish" operator
  def ⟹[A, B, C](f: A ⇒ F[B], g: B ⇒ F[C]): A ⇒ F[C] =
    a ⇒ f(a) // we have an F[B] and a B ⇒ F[C] and we're stuck

}
```

```scala
// A typeclass that describes type constructors that allow composition with ⟹
trait Fishy[F[_]] {

  // Our identity, A ⇒ F[A] for any type A
  def pure[A](a: A): F[A]

  // Composition - the "fish" operator
  def ⟹[A, B, C](f: A ⇒ F[B], g: B ⇒ F[C]): A ⇒ F[C] =
    a ⇒ f(a).flatMap(g) // hey that looks like flatMap!

}
```

```scala
// A typeclass that describes type constructors that allow composition with ⟹
trait Fishy[F[_]] {

  // Our identity, A ⇒ F[A] for any type A
  def pure[A](a: A): F[A]

  // The operation we need if we want to define ⟹
  def flatMap[A, B](fa: F[A])(f: A ⇒ F[B]): F[B]

}
```

Rob Norris explaining that the fish operator (Kleisli composition), can be implemented using flatMap.

Rob Norris @tpolecat

YouTube

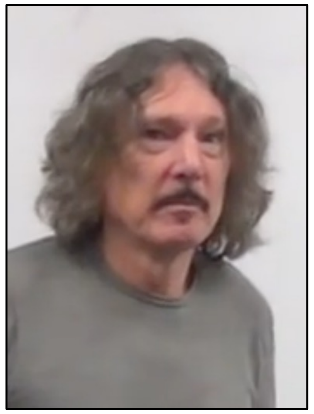
scale.bythebay.io

Rob Norris

Functional Programming with Effects

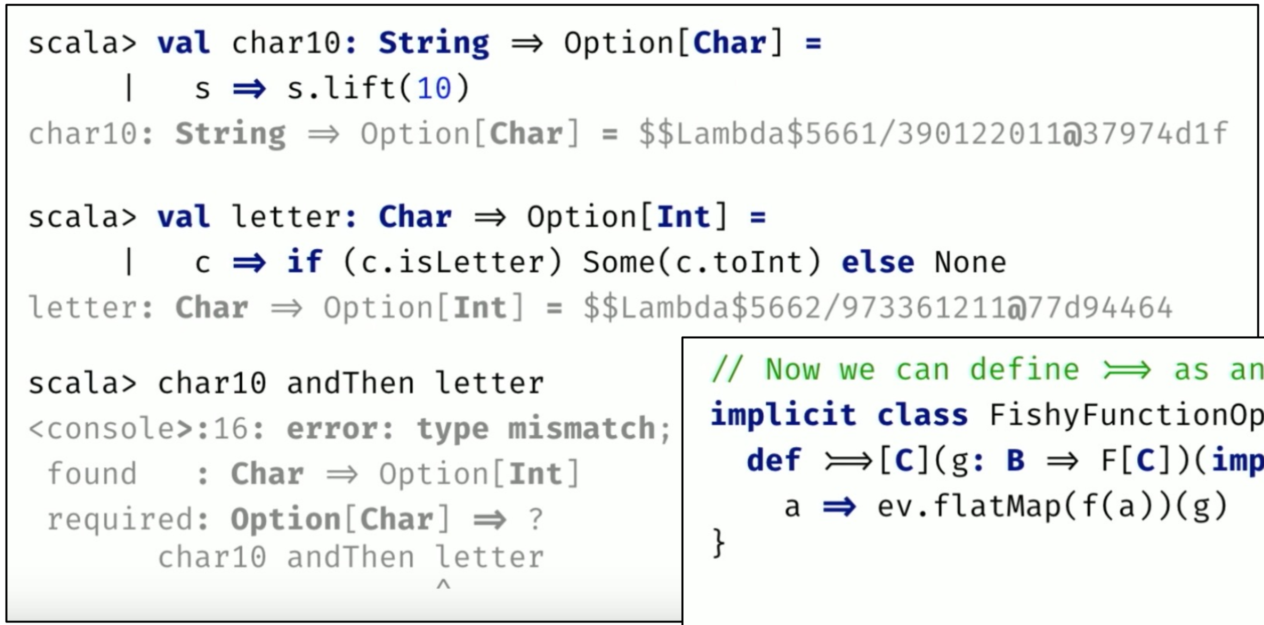
$$f \mathbin{>=>} \underline{g} = \lambda a \rightarrow \text{let } \underline{mb} = f\ a \quad \text{in } mb \mathbin{>>=} g$$

Bartosz Milewski's definition of the fish operator (Kleisli composition) in his lecture on monads.

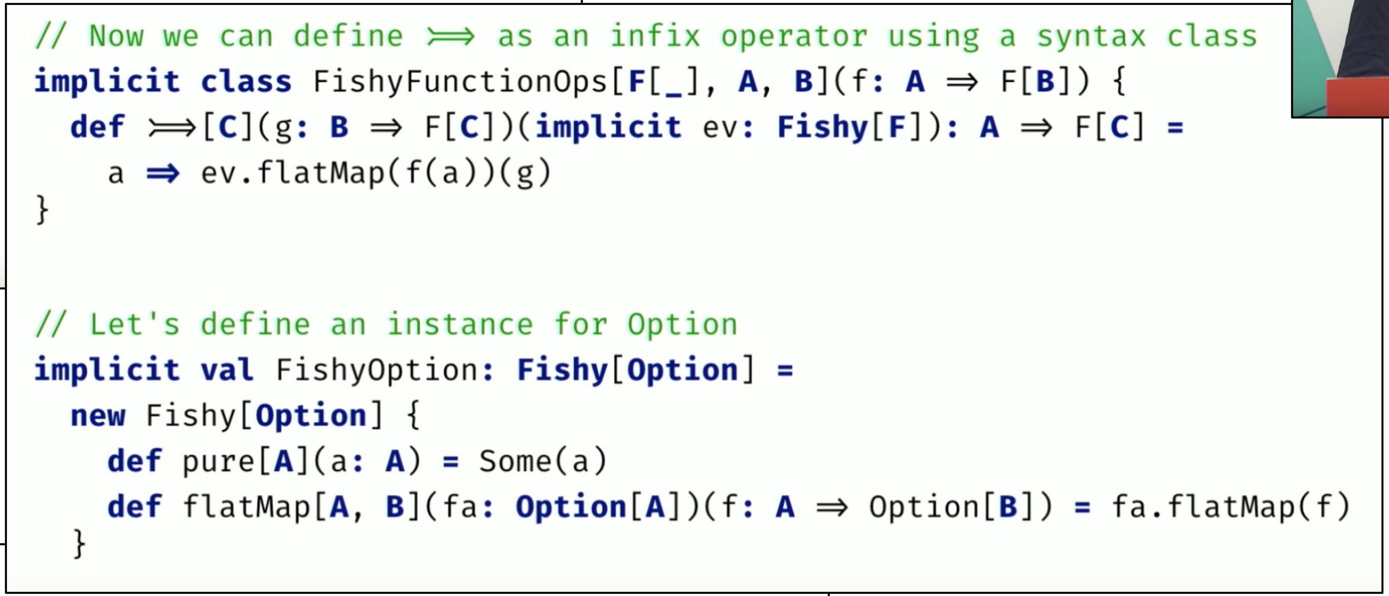


@BartoszMilewski

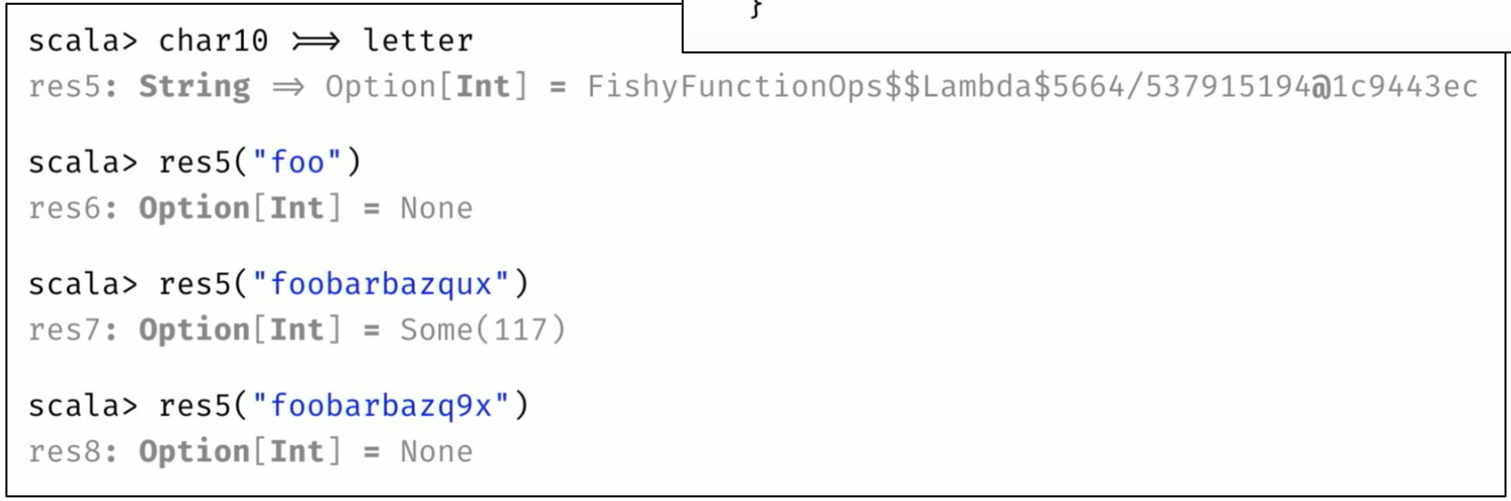
YouTube Category Theory 10.1: Monads



```
scala> val char10: String ⇒ Option[Char] =
     |   s ⇒ s.lift(10)
char10: String ⇒ Option[Char] = $$Lambda$5661/390122011@37974d1f

scala> val letter: Char ⇒ Option[Int] =
     |   c ⇒ if (c.isLetter) Some(c.toInt) else None
letter: Char ⇒ Option[Int] = $$Lambda$5662/973361211@77d94464

scala> char10 andThen letter
<console>:16: error: type mismatch;
 found   : Char ⇒ Option[Int]
 required: Option[Char] ⇒ ?
       char10 andThen letter
                      ^
```



```
// Now we can define ⟹ as an infix operator using a syntax class
implicit class FishyFunctionOps[F[_], A, B](f: A ⇒ F[B]) {
  def ⟹[C](g: B ⇒ F[C])(implicit ev: Fishy[F]): A ⇒ F[C] =
    a ⇒ ev.flatMap(f(a))(g)
}


// Let's define an instance for Option
implicit val FishyOption: Fishy[Option] =
  new Fishy[Option] {
    def pure[A](a: A) = Some(a)
    def flatMap[A, B](fa: Option[A])(f: A ⇒ Option[B]) = fa.flatMap(f)
  }
```



```
scala> char10 ⟹ letter
res5: String ⇒ Option[Int] = FishyFunctionOps$$Lambda$5664/537915194@1c9443ec

scala> res5("foo")
res6: Option[Int] = None

scala> res5("foobarbazqux")
res7: Option[Int] = Some(117)

scala> res5("foobarbazq9x")
res8: Option[Int] = None
```

Rob Norris showing that while we cannot compose effectful functions **char10** and **letter** with normal composition, we are able to compose them using the fish operator (Kleisli Composition).

Rob Norris @tpolecat

YouTube

scale.bythebay.io
Rob Norris
Functional Programming with Effects

Fishy

```scala
// Fishy typeclass
trait Fishy[F[_]] {
  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A ⇒ F[B]): F[B]
}
```

Monad

```scala
// Monad typeclass
trait Monad[F[_]] {
  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A ⇒ F[B]): F[B]
}
```

And this **Fishy** typeclass that we have derived from nothing, using math, is **Monad**. So this scary thing, it just comes naturally and I haven't seen people talk about getting to it from this direction. And so I hope that was helpful.

Rob Norris @tpolecat

```scala
// Abbreviated Definition
sealed trait Option[+A]
case object None extends Option[Nothing]
case class  Some[+A](a: A) extends Option[A]

// Monad instance
implicit val OptionMonad: Monad[Option] =
  new Monad[Option] {
    def pure[A](a: A) = Some(a)
    def flatMap[A, B](fa: Option[A])(f: A ⇒ Option[B]) =
      fa match {
        case Some(a) ⇒ f(a)
        case None    ⇒ None
      }
  }
```

Rob Norris' examples of monad instances, e.g. the ones for **Option**, **Either** and **List**, illustrate the following points by Bartosz Milewski:

- For our limited purposes, a **Kleisli category** has, as objects, the types of the underlying programming language. Morphisms from type A to type B are functions that go from A to a type derived from B using the particular **embellishment**. **Each Kleisli category defines its own way of composing such morphisms, as well as the identity morphisms** with respect to that composition.
- **Since each of the embellishments is different, monadic composition will be implemented differently**, but the overall pattern is the same. It's a very simple pattern: composition that is associative and equipped with identity.

```scala
// Abbreviated Definition
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[S]) extends List[A]

// Monad instance
implicit val ListMonad: Monad[List] =
  new Monad[List] {
    def pure[A](a: A) = a :: Nil
    def flatMap[A, B](fa: List[A])(f: A ⇒ List[B]): List[B] =
      fa.foldRight(List.empty[B])((a, bs) ⇒ f(a) ::: bs)
  }
```

```scala
// Abbreviated Definition
sealed trait Either[+A, +B]
case class Left [+A, +B](a: A) extends Either[A, B]
case class Right[+A, +B](b: A) extends Either[A, B]

// Monad instance
implicit def eitherMonad[L]: Monad[Either[L, ?]] =
  new Monad[Either[L, ?]] {
    def pure[A](a: A) = Right(a)
    def flatMap[A, B](fa: Either[L, A])(f: A ⇒ Either[L, B]) =
      fa match {
        case Left(l)  ⇒ Left(l)
        case Right(a) ⇒ f(a)
      }
  }
```

Rúnar Bjarnason on composing of effectful functions using Kleisli composition, which is defined in terms of flatMap, and for which Scalaz defines the fish operator >=>

```
trait Functor[F[_]] {
  def map[A,B](f: A => B): F[A] => F[B]
}

map(f compose g) = map(f) compose map(g)
map(identity) = identity
```

And there are lots of different kinds of **Functors** like this, but I want to also point out that with functions, **I am really talking about pure functions**.

```
f: A => B
```

If **f** has a side effect, composition is impossible.

Because **composition breaks down if we have side effects**. **It no longer works**. And so what we want to do is we want to **track the effects in the return type of the function. Rather than having side effects**, like returning nulls or throwing exceptions, or something, we are going to **track them in the return type of the function**.

Scala eXchange 2017 Keynote:
Composing Programs

Rúnar Bjarnason
@runarorama

```
f: A => Option[B]
```

Effect: the function **f** might not return any **B**

So here the effect is that the function **f** might not return a **B**, it might return just a **None**.

```
f: A => Option[B]
g: B => Option[C]
```

Problem:
`f andThen g`

But we run into a problem when we have functions of this form, that **we can no longer use regular function composition**. Like we can't say **f andThen g**, if we have both **f** and **g** that return **Option**, because **the types are no longer compatible**.

skills matter **https://skillsmatter.com/skillscasts/10746-keynote-composing-programs**

```
f: A => Option[B]
g: B => Option[C]
```

Solution:
`f andThen (_ flatMap g)`

But **we can solve that just by writing a little more code**. So we can say **f andThen this function that flatMaps g over the result of f**. So we can actually write a composition on these types of functions, that is **not ordinary function composition, it is composition on function and some additional structure**.

```
f: A => Option[B]
g: B => Option[C]

f >=> g : A => Option[C]
```

But we can actually write that as an operator, and in both **Scalaz** and **Cats** it is represented as this sort of fish operator **>=>**.
So if we have **f** that goes from A to Option[B] and **g** that goes from B to Option[C] we have a composite function **f fish g**, that goes from A to Option[C].
And now **this is starting to look like real composition**.

Kleisli Category

- Objects: Scala types
- An arrow from **A** to **B** is a function of type `A => Option[B]`
- Composition: Kleisli composition
  - `f >=> g >=> h = (x => h(x) flatMap g flatMap f)`
  - `identity(x) = Some(x)`

And in fact, **once we have this, we have a category**. And this thing is called a **Kleisli category**, named after a mathematician called Heinrich Kleisli.

So in general we have a **Kleisli category** like this, exactly when the **Functor M** is a **Monad**.

Kleisli Category

- Objects: types **A**, **B**, **F[T]** etc.
- An arrow from **A** to **B** is a function of type `A => M[B]` for some functor **M**.
- Composition: Kleisli composition (`flatMap`)
- Identity: `unit: A => M[A]`

And when we have this kind of thing, we have a **Monad**.

```
trait Monad[M[_]] {
  def flatMap[A,B](h: A => M[B]): M[A] => M[B]
  def unit[A]: A => M[A]
}

flatMap(f >=> g) = flatMap(f) compose flatMap(g)
flatMap(unit) = identity
```

Extracts From Debasish Ghosh's **Functional and Reactive Domain Modeling**

```
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute(m: Market, a: Account): Order => List[Execution]
  def allocate(as: List[Account]): Execution => List[Trade]
}
```

Generalizing this algebra, you have three functions with signatures `A => M[B]`, `B => M[C]`, and `C => M[D]` and **your intention is to compose them together**.

And do you know anything about `M`? In the current example, `M` is a `List`, and `List` **is an effect**, as you saw earlier.

But in functional programming, you always try to generalize your computation structure so that you can get some patterns that can be reused in a broader context.

It so happens that **you can generalize the composition of the three functions over a more generalized effect**, a `Monad` instead of a `List`.

This is achieved through an algebraic structure called a **Kleisli arrow**, that allows functions `f: A => M[B]` and `g: B => M[C]` to compose and yield `A => M[C]`, where `M` is a `Monad`.

It's just like ordinary composition but over effectful functions.

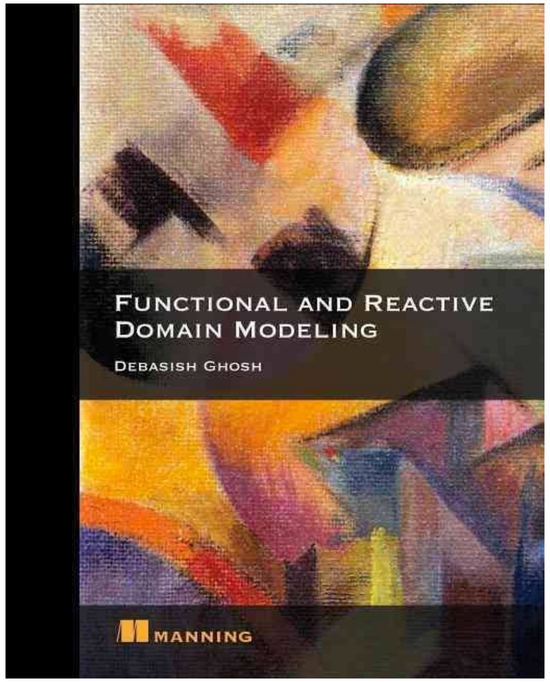
Kleisli is just a wrapper over the function `A => M[B]`.

You can compose two Kleislis if the effect (type constructor) is a monad. Here's part of the definition for `Kleisli` from **Scalaz** (simplified):

```
case class Kleisli[M[_], A, B](run: A => M[B]) {

  def andThen[C](k: Kleisli[M, B, C])(implicit b: Monad[M]) =
    Kleisli[M, A, C]((a: A) => b.bind(this(a))(k.run))

  def compose[C](k: Kleisli[M, C, A])(implicit b: Monad[M]) =
    k andThen this
  //..
}
```

Kleisli just wraps the effectful function. … a computation builds an abstraction without evaluation. Kleisli is such a computation that doesn't evaluate anything until you invoke the underlying function **run**.

This way, using Kleisli, you can build up multiple levels of composition without premature evaluation.



Debasish Ghosh
@debasishg

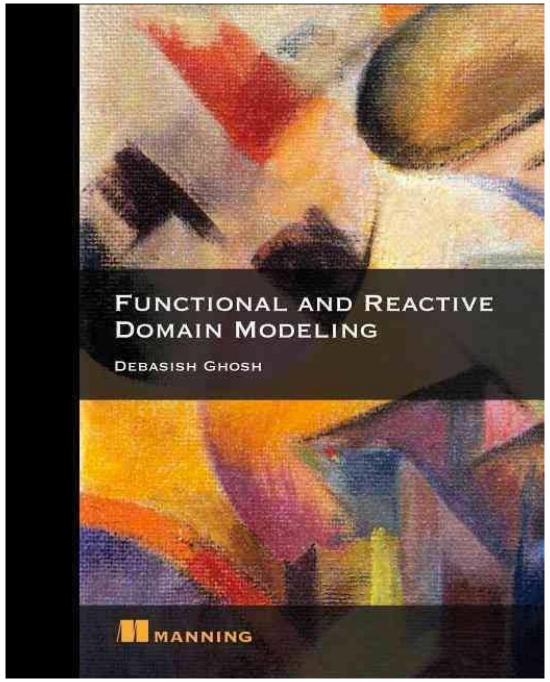
Using **Kleisli composition**, **composing our three functions becomes just function composition with effects**. This also demonstrates how to express your domain algebra in terms of an existing algebraic structure. This is an extremely important concept to understand when talking about algebraic API design. The following listing shows how the domain algebra gets refined with the new pattern that you've just discovered.

```scala
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: ClientOrder => List[Order]
  def execute(m: Market, a: Account): Order => List[Execution]
  def allocate(as: List[Account]): Execution => List[Trade]
}
```

Before
the Kleisli
pattern

```scala
trait Trading[Account, Market, Order, ClientOrder, Execution, Trade] {
  def clientOrders: Kleisli[List, ClientOrder, Order]
  def execute(m: Market, a: Account): Kleisli[List, Order, Execution]
  def allocate(as: List[Account]): Kleisli[List, Execution, Trade]
}
```

Using
the Kleisli
pattern



4.4.3 Final composition—follow the types

Now you can use the three functions to build a larger domain behavior of trade generation. Just follow the types and you have the complete trade-generation process (simplified for demonstration), from client orders to the allocation of trades to client accounts.

```scala
def tradeGeneration(market: Market, broker: Account, clientAccounts: List[Account]) = {
  clientOrders andThen
    execute(market, broker) andThen
      allocate(clientAccounts)
}
```

…Kleisli gives you a computation for which you need to invoke the underlying [run] function for evaluation… run the preceding `tradeGeneration` function for a market, a broker account, and a list of client accounts.

Debasish Ghosh
@debasishg

Sample effectful functions f, g and h, to be composed using Kleisli composition

```scala
// sample A => M[B] function
val f: String => List[String] = _.split(" ").toList.map(_.capitalize)
assert(f("dog cat rabbit parrot") == List("Dog", "Cat", "Rabbit", "Parrot"))

// sample B => M[C] function
val g: String => List[Int] = _.toList.map(_.toInt)
assert(g("Cat") == List(67, 97, 116))

// sample C => M[D] function
val h: Int => List[Char] = _.toString.toList
assert(h(102) == List('1', '0', '2'))

// sample A value
val a = "dog cat rabbit parrot"
// sample B value
val b = "Cat"
// sample C value
val c = 67
// sample D value
val d = '6'

/* expected value of applying f >=> g >=> h to "dog cat rabbit parrot" */
val expected =  List(
  /* Dog    */ '6', '8', '1', '1', '1', '1', '0', '3',
  /* Cat    */ '6', '7', '9', '7', '1', '1', '6',
  /* Rabbit */ '8', '2', '9', '7', '9', '8', '9', '8', '1', '0', '5', '1', '1', '6',
  /* Parrot */ '8', '0', '9', '7', '1', '1', '4', '1', '1', '4', '1', '1', '1', '1', '1', '6')
```

In functional programming we take an interesting interpretation of **List**, we sometimes think about it as a kind of **non-determinism**, we can define functions that might return **any number of answers**.

Rob Norris @tpolecat

Say you have any type **A** and you'd like to add **the capability of aggregation**, so that you can treat a collection of **A** as a separate type. You do this by constructing a type **List[A]** (for which the corresponding type constructor is **List**), which adds **the effect of aggregation** on **A**.

Debasish Ghosh

@debasishg

Experimenting with the Kleisli composition of f, g and h

using Scala facilities

```scala
{ val fgh: String => List[Char] = f(_) flatMap g flatMap h
  val result: List[Char] = fgh(a)
  assert( result == expected ) }

{ val result: List[Char] = f(a) flatMap g flatMap h
  assert( result == expected ) }

{ val fgh = f andThen (_ flatMap g flatMap h)
  val result: List[Char] = fgh(a)
  assert( result == expected ) }

{ val result: List[Char] = for {
    b:String <- f(a)
    c:Int    <- g(b)
    d:Char   <- h(c)
  } yield d
  assert( result == expected ) }

{ val result: List[Char] =
               f(a) flatMap {
    b:String => g(b) flatMap {
    c:Int =>    h(c) map {
    d:Char =>   d    }}}
  assert( result == expected ) }

{ val result: List[Char] =
               f(a) flatMap {
    b:String => g(b) flatMap {
    c:Int    => h(c) }}
  assert( result == expected ) }
```

using Scalaz operators

```scala
import scalaz._
import Scalaz._
import Kleisli._

val fk: Kleisli[List,String,String] = kleisli(f)
val gk: Kleisli[List,String,Int] = kleisli(g)
val hk: Kleisli[List,Int,Char] = kleisli(h)

assert(fk.run.isInstanceOf[String => List[String]])
assert(gk.run.isInstanceOf[String => List[Int]])
assert(hk.run.isInstanceOf[Int => List[Char]])
assert(fk.run(a) == f(a))
assert(gk.run(b) == g(b))
assert(hk.run(c) == h(c))

for (fghk: Kleisli[List,String,Char] <- List(

  // composing existing Kleislis
  fk >=> gk >=> hk,                                  // fish operator
  fk andThen gk andThen hk,                          // andThen alias for >=>

  // composing Kleislis created on the fly
  kleisli(f) >=> kleisli(g) >=> kleisli(h),          // fish operator
  kleisli(f) andThen kleisli(g) andThen kleisli(h),  // andThen alias for >=>

  // less verbose way of composing Kleislis created on the fly
  kleisli(f) >==> g >==> h,                          // a bigger fish ;-)
  kleisli(f) andThenK g andThenK h                   // andThenK alias for >==>

)) { assert(fghk.run(a) == expected) }
```

Comparing some ways of composing f, g and h

Using Scala's flatMap and for

```scala
val fgh = (a:String) => f(a) flatMap g flatMap h

val result = fgh(a)
```

```scala
val fgh = f andThen (_ flatMap g flatMap h)

val result = fgh(a)
```

```scala
val fgh = (a: String) =>
  for {
    b <- f(a)
    c <- g(b)
    d <- h(c)
  } yield d

val result = fgh(a)
```

using Scalaz operators >=> and >==>

```scala
val fghk = kleisli(f) >=> kleisli(g) >=> kleisli(h)

val result = fghk.run(a)
```

```scala
val fghk = kleisli(f) >==> g >==> h

val result = fghk.run(a)
```

```scala
val fk = kleisli(f)
val gk = kleisli(g)
val hk = kleisli(h)

val fghk = fk >=> gk >=> hk

val result = fghk.run(a)
```

Example of composing functions that return Option

```scala
import scalaz._
import Scalaz._
import Kleisli._

case class Car(insurance:Option[String])
case class Driver(car:Option[Car])
case class Company(driver:Option[Driver])

val driver = (company:Company) => company.driver
val car = (driver:Driver) => driver.car
val insurance = (car:Car) => car.insurance

val driverK = kleisli(driver)
val carK = kleisli(car)
val insuranceK = kleisli(insurance)
```

```scala
for (insuranceName <- List(

  (c:Company) => driver(c) flatMap car flatMap insurance,

  (driver andThen ( _ flatMap car flatMap insurance)),

  (company:Company) => for {
    driver <- company.driver
    car <- driver.car
    insurance <- car.insurance
  } yield insurance,

  kleisli(driver) andThen kleisli(car) andThen kleisli(insurance) run,

  kleisli(driver) andThenK car andThenK insurance run,

  kleisli(driver) >=> kleisli(car) >=> kleisli(insurance) run,

  kleisli(driver) >==> car >==> insurance run,

  driverK >=> carK >=> insuranceK run

)){
  val noDriver = Company(driver = None)
  val noCar = Company(Some(Driver(car = None)))
  val uninsured = Company(Some(Driver(Some(Car(insurance = None)))))
  val insured = Company(Some(Driver(Some(Car(insurance = Some("Axa"))))))
  assert(insuranceName(noDriver) == None)
  assert(insuranceName(noCar) == None)
  assert(insuranceName(uninsured) == None)
  assert(insuranceName(insured) == Some("Axa"))
}
```