Compositionality and Category Theory

A montage of slides/transcript for sections of the Scala eXchange 2017 closing keynote: "Composing Programs"

keynote speaker: **Rúnar Bjarnason** - https://twitter.com/**runarorama**

video of keynote: https://skillsmatter.com/skillscasts/10746-keynote-composing-programs

get free access to the keynote (and all other videos!) simply by registering on the skillsmatter site

transcript/montage by https://twitter.com/**philip_schwarz**

Compositionality is this property that we construct software from modular pieces that we can individually understand, and then **if we understand the pieces and we understand how those pieces are put together then we automatically understand the whole system**.

There is nothing outside of the understanding of the pieces and the composition of our understanding that will inform how we understand the whole.

So the definition of compositionality that I like is this:

Software is *compositional* to the extent that we can understand the **whole** by understanding the **parts** and the rules of **composition**.

rúnar bjarnason @runarorama

Therefore, a compositional expression, or a compositional program is going to be a nested expression, it is going to be some nested structure, and each subexpression is going to mean something, it is going to be a meaningful program on its own, and then the meaning of the whole thing is going to be composed of the meaning of the subexpressions, and this is going to work at every level:

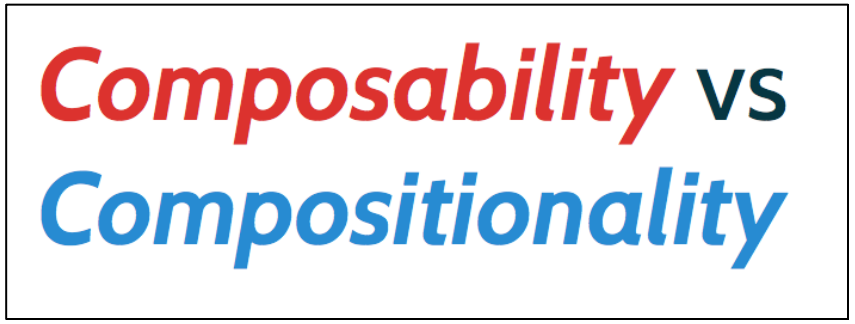
- A *compositional* expression is a nested structure.
- Each expression *means* something.
- The meaning is *composed* of the meanings of the subexpressions.

A concise way of saying that is:

The **composition** of the **meanings** is the **meaning** of the **composition**.

There is a pun in here on composition. So there are **two notions of composition**. There is composition in the **expression space** and then there is composition in the **meaning space**. In the **structure** or **interpretation** spaces. Or in **syntax** and **semantics**, if you will. And **compositionality in one should mirror the compositionality in the other**. They should be structurally similar in some way.

Note that I am not talking about composability. **Composability** conveys **a weaker notion than** **compositionality**.



Composability vs
Compositionality

rúnar bjarnason @runarorama

Composability conveys that something is **able to be composed**. Like if you twist its arm. But **compositional expressions are natively and fractally composable**.

In fact I want to say that **compositional** **expressions do nothing other than be composed**. That is all they do.

```
val fr = new FileReader("thefile.txt")
val br = new BufferedReader(fr)

var line = br.readLine()

var count = 0

while (line != null) {
  val words = line.split("\\s")
  for (w <- words) {
    count += 1
  }
  line = br.readLine()
}

br.close()

println(count)
```

So here is a program that I wrote. Not my best program. It is ok.

So what this does is count the number of words in a file. It is **a perfectly fine program, except it is not compositional**.

The individual parts of this are not complete in and of themselves.

You need to read the whole program in order to understand what any of the parts are doing.

And **related parts are very far apart**, like opening the buffered reader and closing it are very far apart **and there is no sort of connection** between the two.

rúnar bjarnason @runarorama

I want to **contrast** this with a program that I wrote that does the same thing but does so in a **compositional** way.

```
io.linesR("thefile.txt")
  .flatMap(s => emits(s.split("\\s")))
  .map(_ => 1)
  .fold(0)(_ + _)
  .to(stdout)
```

So this program is written using the FS2 library, a functional streams library for Scala.

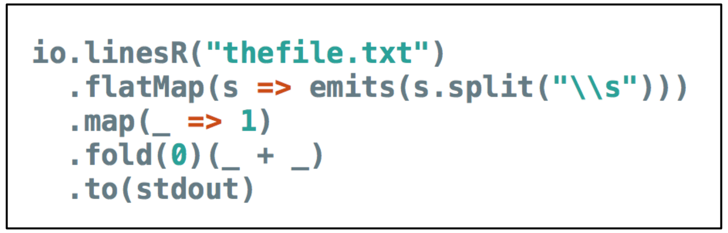
So here the logic of the expression is much clearer, but not only that: **every part of this is meaningful**.

- For instance, the first line is a stream of all the lines in a file. So it has that meaning.
- And **that meaning is independent of the next thing, which also has its own meaning**.
- So this flatMap expression here means that for every string in the incoming stream I am going to split it on word boundaries, on whitespace, and then I am going to emit, as a stream, the words that I see. So **this expression, no matter what came before or after, is going to take a stream of strings** and turn it into a stream of words in those strings.
- And then the map part is going to **turn all of the incoming elements in any stream into ones**, it is going to replace all the elements with the number one.
- The fold expression is going to **sum any stream of numbers**
- And the .to is going to take any stream of strings and it is going to send it to the standard output to be printed onto the screen, **no matter what came before**.

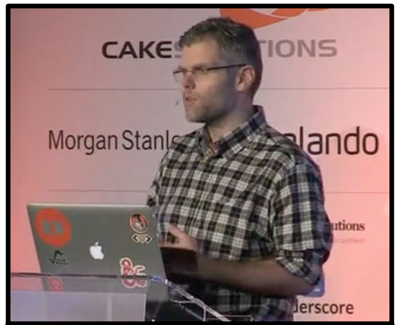
And then **the composition of those meanings is the meaning of the whole program**.

Every part is a meaningful expression and the whole thing has the same character as the parts.

We can continue reusing this as a part of a larger stream processing system. So this is a honest to goodness stream.



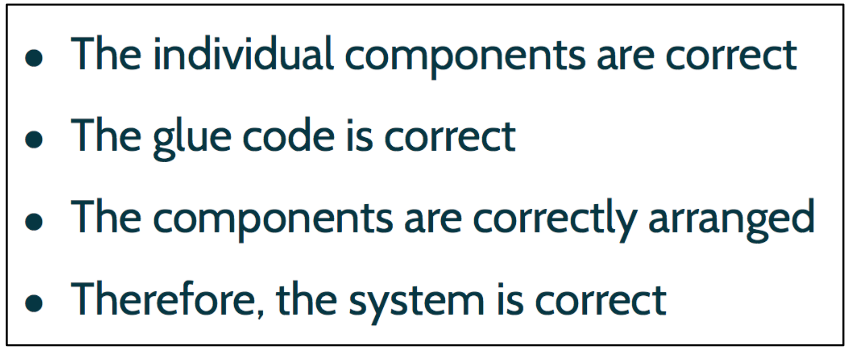
```
io.linesR("thefile.txt")
  .flatMap(s => emits(s.split("\\s")))
  .map(_ => 1)
  .fold(0)(_ + _)
  .to(stdout)
```



rúnar bjarnason @runarorama

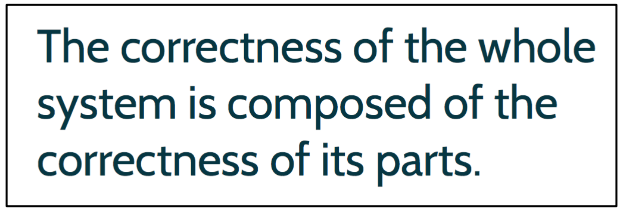
Another thing about this is that it is **very easy to verify**. The logic is **very clear**.

But also, if the individual elements of this are correctly written and the individual parts of this are correctly written and we have sort of chosen the right parts in the right order then the whole thing is correct.



- The individual components are correct
- The glue code is correct
- The components are correctly arranged
- Therefore, the system is correct

That is the **correctness of the whole is in some sense composed of the correctness of the parts**.



The correctness of the whole system is composed of the correctness of its parts.

And I fact **when we are resoning about really large scale systems we have no hope of reasoning about their correctness, whether they work, without this kind of compositional reasoning**.

```
io.linesR("thefile.txt")
  .flatMap(s => emits(s.split("\\s")))
  .map(_ => 1)
  .fold(0)(_ + _)
  .to(stdout)
```

The program that we had earlier, we can pull it apart into individual components so we can give each component a name each of which is meaningful on its own.

```
val lines = io.linesR("thefile.txt")
val words = _.flatMap(s => emits(s.split("\\s")))
val ones = _.map(_ => 1)
val sum = _.fold(0)(_ + _)
val print = _.to(stdout)

val prg = print(sum(ones(words(lines))))
```

So components can be reused. So because this is compositional, it is naturally modular
(It is not necessarily the other way around, that modular software is compositional).

So this final program here is literally a composition of the functions print, sum, ones, words and just applied to the lines value.

And we can write that explicitly as a composition like this:

```
val f = print
  compose sum
  compose ones
  compose words

val prg = f(lines)
```

And ideally we want all of our programs to be this way: a single function applied to some input

This is possible because functions are compositional: you can always compose functions as long as the types line up.

Functions are compositional

rúnar bjarnason @runarorama

And really I want to say that functional programming as such is really the study of compositional software.

Functional programming is really the study of *compositional software*

Functional Programming is **taking this idea of compositionality to its logical conclusion**. Taking that idea seriously and studying the implications of it.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "\text{g compose f}"'] & B \arrow[d, "g"] \\
 & C
\end{tikzcd}$$

`(x:A) => g(f(x))`

Functions are **compositional** in that if we have a function **f** that takes an **A** and returns a **B** and we have a function **g** that takes a **B** and returns a **C** then we have a **composite** function **g compose f** that does both **f** and **g** and the implementation is lambda of **x**, **g** of **f** of **x**.

So both of the paths through this diagram are the same function: **f** followed by **g** and **g compose f**.

rúnar bjarnason @runarorama

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[rr, bend left, "\text{g compose f}"] \arrow[rrr, bend left=60, "\text{h compose g compose f}"] & B \arrow[r, "g"'] \arrow[rr, bend right, "\text{h compose g}"'] & C \arrow[r, "h"'] & D
\end{tikzcd}$$

And **composition is associative**, so it doesn't matter if we do **f** followed by **h compose g** or if we **do g compose f** followed by **h**. Any two of those paths through this graph are going to be the same function. They are going to be **f** followed by **g** followed by **h**.
So **if we have any three functions we can compose them and it doesn't matter the order in which we compose them**. They don't commute past each other: we can't change the sequence of the functions because the types have to line up.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "f"'] & B \arrow[d, "b \Rightarrow b"] \\
 & B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "a \Rightarrow a"] \arrow[dr, "f"'] & A \arrow[d, "f"] \\
 & B
\end{tikzcd}$$

And then for every type there is going to be an **identity function** on that type that does nothing, and what it means to do nothing is precisely that when composed with another function **f** you are going to get the same function you started with. So the identity function is a sort of no-op.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "g \text{ compose } f"'] & B \arrow[d, "g"] \\
 & C
\end{tikzcd}$$

`(x:A) => g(f(x))`

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[rr, bend left, "g \text{ compose } f"] \arrow[rrr, bend left=50, "h \text{ compose } g \text{ compose } f"] & B \arrow[r, "g"'] \arrow[rr, bend right, "h \text{ compose } g"'] & C \arrow[r, "h"'] & D
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "f"'] & B \arrow[d, "b \Rightarrow b"] \\
 & B
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r, "a \Rightarrow a"] \arrow[dr, "f"'] & A \arrow[d, "f"] \\
 & B
\end{tikzcd}$$

Category

- Objects
- Arrows between objects
- Composition of arrows
  - Which is associative
  - And has an identity

So whenever we have this structure, we have a thing called a **category**.

Whenever we have some **objects**, we have some **arrows** between those objects, we have **composition** on the arrows, which is **associative** and has an **identity**. And this is really all of category theory. Here on this slide.

I mean it is a little bit vague but essentially this is all of it. So everyone now knows category theory.

rúnar bjarnason @runarorama

Scala forms a Category

- Objects: Scala types
- Arrows: Scala functions
- Composition: function composition
  - `f compose g compose h = (x => f(g(h(x))))`
  - `identity = (x => x)`

Scala actually forms a **category** in a way that we saw previously. The **objects** in this category are the **Scala types**.

The **arrows** between the objects are Scala **functions that go from one type to another**, and then the **composition** of the arrows is going to be **function composition**. So whenever we have an arrow from **A** to **B** and an arrow from **B** to **C** we have a **composite arrow** from **A** to **C**, and this works for any arrows and any objects **A**, **B** and **C**.

And then there is an **identity arrow** which is the **identity function**, and then **function composition** is **associative** because it is always implemented as lambda x, f of g of h of x, no matter how we associate the parentheses.

Scala forms a Category

- Objects: Scala types
- Arrows: Scala functions
- Composition: function composition
  - `f compose g compose h = (x => f(g(h(x))))`
  - `identity = (x => x)`

Scala forms a **category**, and what is **category theory** really about?
Category theory is actually just the abstract study of composition as such. So it is asking the question 'what kind of stuff is compositional?', and studying those things in the abstract. What kind of things can we say about stuff that is compositional?

Category Theory is really the abstract study of *composition*

rúnar bjarnason @runarorama

```scala
trait Monoid[M] {
  def empty: M
  def append(m1: M, m2: M): M
}
```

Another simple example of a compositional thing is a **Monoid**. A Monoid is **a special case of a category**. So in Scala we define a Monoid on some type **M** and it has **two methods**, or two values in it. One of them is a function that takes two **M**s and returns one **M** and that should be **an associative operation**. So it should be able to take two of these **M**s and **put them together**, **append one to the other**. And then **empty** should be some element of some value type **M** that **does nothing**. It is the **unit** for this **append**. So empty appended with anything else is that same thing, on either side.

Monoid

1. A type
2. An associative binary operation
3. An identity element for that operation

Examples

- `Int` with (+, `0`)
- `Int` with (*, `1`)
- `Boolean` with (&&, `true`)
- `String` with (++, `""`)
- `A => A` with (`compose`, `identity[A]`)

A **Monoid** is given by some **type** in Scala, some **associative binary operation**, an **identity element** for that operation. Some examples of this are

- the **integers** with identity element zero,
- the **integers** with multiplication and 1, so 1 multiplied by anything else is that same thing
- **booleans** with && and true, as well as || and false
- **strings** with string concatenation, where the string identity element is the empty string
- **functions** from **A** to **A** for some type **A**, called **endofunctions**, or **endomorphisms** on **A**, **endo** means within, and the operation on that is just **function composition**, and the **identity** is the identity function on **A.**

All of these are monoids.

A monoid is a category with one object

- Objects: The type `M`
- Arrows: Values of type `M`
- Composition: `append`
- Identity: `empty`

Another way of defining a Monoid is to say that it is a category with one object.

So we have seen what categories are. Now an example of a category is a Monoid. A Monoid is a category with just one object.

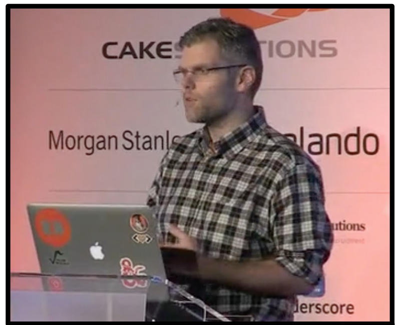
So we can think of the Monoid as having just one object which we can imagine being the type **M**, but actually it doesn't matter what is inside of this object. It is totally opaque. The only thing we care about are the values of type **M**, which are going to be the **arrows**. Arrows are going to go from that object to itself.

And the **composition on the arrows**, **notice the arrows here are not functions, they are values**, and the composition on these values is going to be the **append** on the Monoid, and the identity is going to be the **empty element** of the Monoid.

And so this forms an honest to goodness category that has one object that just works to anchor all of the arrows.

$$\begin{tikzcd}
\ast \arrow[r, "x"] \arrow[dr, "y \cdot x"'] & \ast \arrow[d, "y"] \\
& \ast
\end{tikzcd}$$

And so whenever we have an element **x** and an element **y** of the Monoid, we have a **composite element y compose x**, or **y times x**, or **y plus x**, or whatever the operation is, and I have denoted the one object as being star, because I don't actually care what the structure of that is, **the only thing that really matters are the arrows**, and the directionality of the arrows matters precisely because I can can't swap **x** and **y**, whenever I am talking about **y compose x**, I am talking about doing **x** and then **y**, in that order.



rúnar bjarnason @runarorama

$$\begin{tikzcd}
\ast \arrow[r, "x"] \arrow[dr, "x"'] & \ast \arrow[d, "1"] \\
& \ast
\end{tikzcd}$$

$$\begin{tikzcd}
\ast \arrow[r, "1"] \arrow[dr, "x"'] & \ast \arrow[d, "x"] \\
& \ast
\end{tikzcd}$$

And then every Monoid is going to have an **identity arrow**, which is going to be the identity element in the Monoid. For example, in the integer multiplication it is going to be the number one.

$$\begin{tikzcd}
\ast \arrow[r, "x"'] \arrow[rr, bend left, "y \cdot x"] \arrow[rrr, bend left=60, "z \cdot y \cdot x"] & \ast \arrow[r, "y"'] \arrow[rr, bend right, "z \cdot y"'] & \ast \arrow[r, "z"'] & \ast
\end{tikzcd}$$

And then **composition** in a Monoid is **associative**, as it is in any other category, that is if we have the expression **z** + **y** + **x**, it doesn't matter if we consider it to be **y** + **x** and then adding **z** on the front or whether we say **z** + **y** and then adding **x** on the back.

Compositional reasoning

```
append(wc(s1), wc(s2)) == wc(s1 ++ s2)

append(wc(s), wc("")) == wc(s)
append(wc(""), wc(s)) == wc(s)
```

This is an example of using **compositional reasoning**. We are reasoning that the **concatenation** of the **word counts** of the strings should be the same as the **word count** of the concatenated strings. And we want to also preserve that if we **count the words** in a string and then append the **word count** of the **empty string**, that should do nothing.

Monoid homomorphism

```
append(wc(s1), wc(s2)) == wc(s1 ++ s2)

append(wc(s), wc("")) == wc(s)
append(wc(""), wc(s)) == wc(s)
```

So this is an example of a **Monoid homomorphism**. It is **a function, from one Monoid to another, that preserves the structure**.

Homomorphism, from Greek, homo means same, and morphe means form. So it preserves the structure, or the form of the Monoid.

The concatenation of the word counts is the word count of the concatenation.

rúnar bjarnason @runarorama

Monoid homomorphism

```
s1.length + s2.length == (s1 ++ s2).length

"".length == 0
```

Another example of a **Monoid homomorphism** is taking the length of a string. So if I concatenate the length of s1 and s2 that should be the same as the length of the combined string, s1 plus s2. And then the length of the empty string should be zero.
So **the length of the composition is the composition of the lengths**. That's another example of **Monoid homomorphism**.

Monoid isomorphism

```
!(x || y) = !x && !y
!true = false

!(x && y) = !x || !y
!false = true
```

If I have a homomorphism going both ways then I have a thing called a Monoid isomorphism.
So here, **not (x or y)** is the same as **(not x) and (not y)**. So **the inverse of the combination is the combination of the inverses**.

There are two different Monoids on the booleans and there is a homomorphism going one way and there is a homomorphism going the other way, and so **they form an isomorphism**, so they have the same structure in a sense.

Category Theory is really the abstract study of ***homomorphisms***

And really I want to say that **category theory is actually about studying homomorphisms**. **That is the plot of category theory**. It is about studying composition and compositional things but the things that we want to say about them is how do we compare those structures. How is structure preserved across mappings. How do we reason compositionally about stuff.

The category *Mon* of monoids

- Objects: monoids
- Arrows: monoid homomorphisms
- Composition: function composition

All of the Monoids, taken together, form a category, where the objects are the Monoids and the arrows are the Monoid homomorphisms and there is composition on those, which is just function composition, but it has this interesting property that **if one map preserves a Monoid structure and then the next map also preserves a Monoid structure then the composite map preserves that structure as well**.

rúnar bjarnason @runarorama

The category *Cat* of categories

- Objects: categories
- Arrows: ***category homomorphisms***
- Composition: ?

And since Monoids are categories, we can actually generalize this and say **there is not just a category of Monoids**, **there is a whole category of categories**, where the objects are categories and the arrows are **category homomorphisms**. And **this is the real plot of category theory**.

The category *Cat* of categories

- Objects: categories
- Arrows: ***functors***
- Composition: functor composition

So what kind of things are **category homomorphisms**? They are **Functors**. So it is going to go from one category to another, and it is going to preserve the structure of the category.

Functor

F: C → D

- Takes every object in C to an object in D
- Takes every arrow in C to an arrow in D
- Composition and identity are preserved

A **Functor** from a category **C** to a category **D** is going to take every object in category **C** and it is going to give you an object in **D**. And it is going to **take every arrow in C to an arrow in D that corresponds with it**, in such a way that **composition and identity are preserved**, that is, this is a **homomorphism**, it needs to be that.

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

So in **Scala** we can talk about **Functors**, and when we talk about them in Scala we are actually talking about Functors from the Scala category, with types and functions, to itself. And these are called **endofunctors**. Again, endo means within. And so **this Functor is going to take a type T and it is going to turn it into a type F[T].** So it is type construction that needs an additional type. And then on the arrows, on the functions, it is going to **take any arrow h** that goes from **A** to **B**, and it is going to **turn it into an arrow** that goes from **F[A]** to **F[B].**

For instance, if **F** is **List**, then **Functor** is going to take a function on the elements of the **list** and turn it into a function that applies that function to all of the elements of the **list**.

```scala
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 that has to be a **homomorphism**. So the Functor laws are the homomorphism law and we say that **mapping a composite function should be the same as composing the maps**.

So map of **f** compose **g** should be map **f** compose map **g**. That's the Functor law. And also that the map of the identity should be identity. It should preserve the structure of the category across this mapping. And that is true for all Functors.

rúnar bjarnason @runarorama

```scala
implicit val optionF = new Functor[Option] {
  def map[A,B](f: A => B): Option[A] => Option[B] =
    { case Some(a) => Some(f(a))
      case None => None
    }
}

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

Just to give **an example of a Functor implemented in Scala**, I have **Option** here, so this is going to preserve the sort of **Option**-ness, or the **None**-ness of, if we have a **None**, in our incoming **Option**[**A**], that's going to be preserved, and if we have **Some** in our **Option** A, that is also going to be preserved. It is just going to apply the function **f** to whatever **a** was inside of there.

```
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**.

rúnar bjarnason @runarorama

`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**.

`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**.

```
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**.

rúnar bjarnason @runarorama

```
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.

- The objects in this category are the ordinary Scala types, but in this category now, an arrow from **A** to **B** is not a function from **A** to **B**, it is a function from **A** to **Option[B]**, when we are talking about the **Kleisli category** for **Option**.
- And then **composition** is not going to be ordinary function composition, it is going to be this **special fish operator**, it is going to be our **Kleisli composition**. So **f compose g compose h** is going to be implemented as lambda **x**, **h** of **x** and then **flatMap g** and then **flatMap f**, so it is going to be like a for comprehension, essentially.
- And the **identity** arrow for these Kleisli arrows is not going to be the identity function, because we need to return an **Option**, but what we are going to do is return the **do nothing Option**, i.e. the **Some**. The **effect** of **Option** is the **None**, the **effect** of the **Option** type is to **not return a value**, but in the identity here we are actually **not going to have the effect**, we are going to return some **x**.

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)`

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

rúnar bjarnason @runarorama

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]`

So in a Kleisli category in general the objects normally are some types, Scala types, and then an arrow, instead of being from A to B goes from A to M[B] for some Functor M. And then composition is going to be Kleisli composition, which is going to be implemented by flatMap. And our identity is going to be, usually called unit or pure.
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
```

And we can think about a Monad in Scala, in a Scala category, we can think of it as being a Functor from a Kleisli category into the regular Scala category.
So you can see that **it takes every type A to another type M[A],** but then **it takes every arrow in the Kleisli category**, that goes from A to M[B] (so that's a Kleisli arrow), it takes that **to the regular Scala category, to an arrow of type M[A] => M[B].**
And the Monad laws follow from the fact that this is a Functor, that this is a category homomorphism.
So this is saying that **flatMap of f compose g** is going to be the same as **flatMap f composed with flatMap g**. And then flatMapping of the **unit**, which is the **identity** arrow in the Kleisli category, that's going to be the identity arrow in the Scala category.
So we can reason about Monads using homomorphisms as well.

So Monads essentially allow us to recover composition when we are talking about effectful functions

And there are lots of these, lots of Monads.

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

Effect: the function **f** might return zero or many **B**

So here with lists the **effect** might be the function returns zero or many **B**s instead of just one.

```
f: A => (B,C)
```

Effect: the function **f** returns an extra **C** in addition to the **B**

And then if we have a Monoid on some **C** then we can return an extra **C**, or some log on the side, and then the **composition** is going to take care of the concatenation of those **C**s for us.

```
f: A => (R => B)
```

Effect: the function **f** will request an additional value of type **R**

We might want to have the **effect** that the function requests an additional value of type **R** that it is going to use to produce its **B**. This is called the **Reader Monad**.

```
f: A => (S => (B,S))
```

Effect: the function **f** will keep some state of type **S** which it may use to construct the **B**

Another **effect** we might want to do is have some kind of **state**, and so in this type, this is an arrow from **A** to **B** in the **Kleisli category** for state, but the effect that we have is that we are going to **keep some state** of type **S** across this as well, that we may use to construct the **B** as well.

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

Effect: the function **f** requests some I/O effect before it can produce **B**

And then we can do arbitrary **effects**, using things like **IO**, or **Task**, or **Future**, if you want to do things like **asynchronous requests**, where the effect is that the function is going to request some **IO** to happen, it is not actually going to do it, but it is going to request of whoever receives that **IO** thing, that it needs to be done.

rúnar bjarnason @runarorama

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

The monad **M** describes the effect of **f**

And so in general a Monad like this describes some **effect of a function** and the upshot of this is that **we can compose effects**