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**.

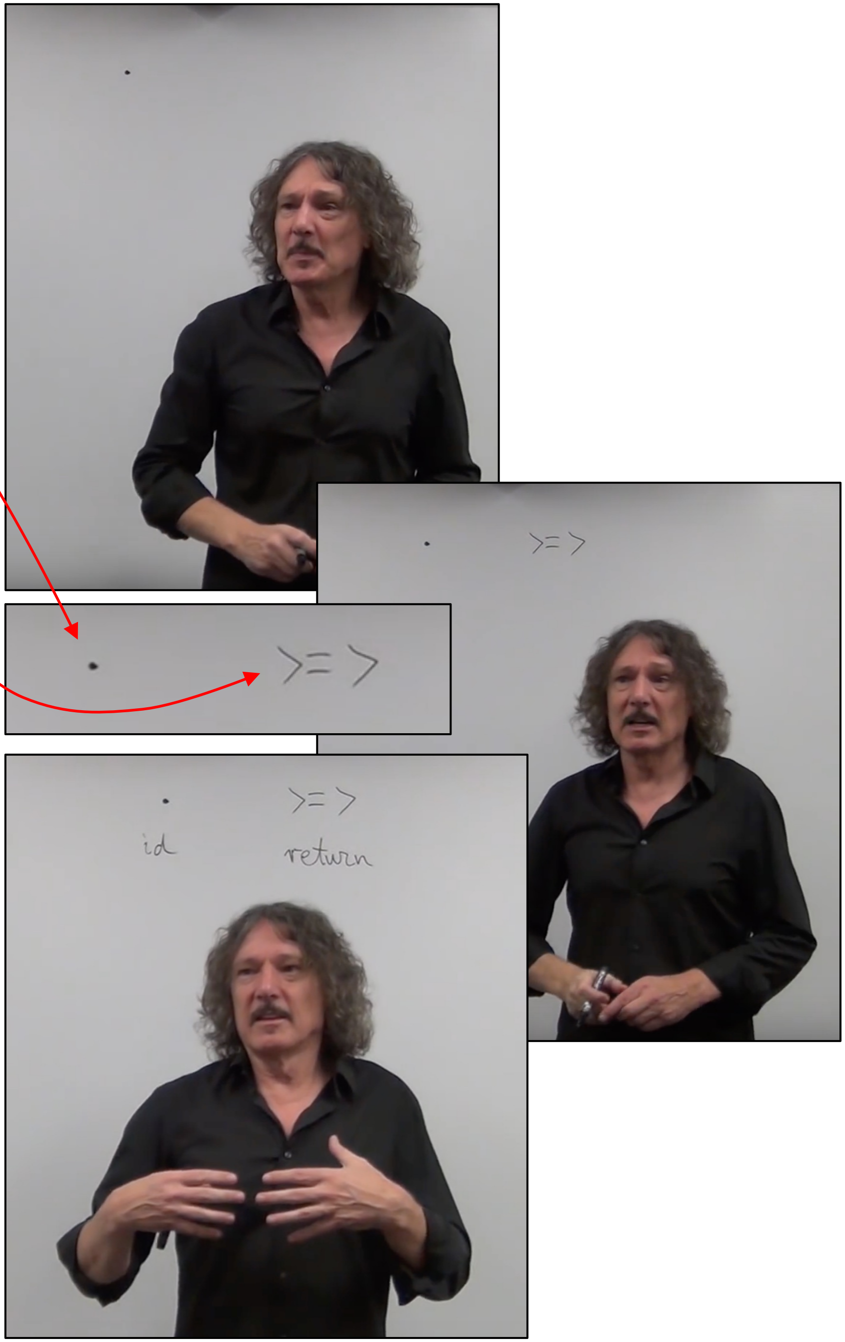
And the power of functions really is in the dot. That's where the power sits. **Dot is the composition operator** in **Haskell**.
It combines two functions so the output of one function becomes the input of the other.
So that explains what functions are really, **functions are about composition.**

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 functons 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

Let's recap, using material by Rob Norris, why we need Kleisli composition (the fish operator), which is defined in terms of flatMap (aka bind)

Rob Norris
@tpolecat

So what about ...

- Partiality?
- Exceptions?
- Nondeterminism?
- Dependency injection?
- Logging?
- Mutable state?
- Imperative programming generally?

Six Effects

What do they have in common?

- All compute an "answer"but also encapsulate something extra about the computation.
- This is what we call an effect. But it's very vague. Can we be more precise about what they have in common?

All have shape F[A]

```scala
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`.

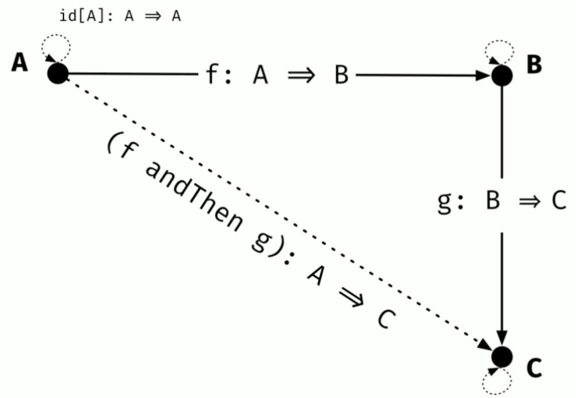
scale.bythebay.io: Rob Norris, Functional Programming with Effects

But they don't compose!

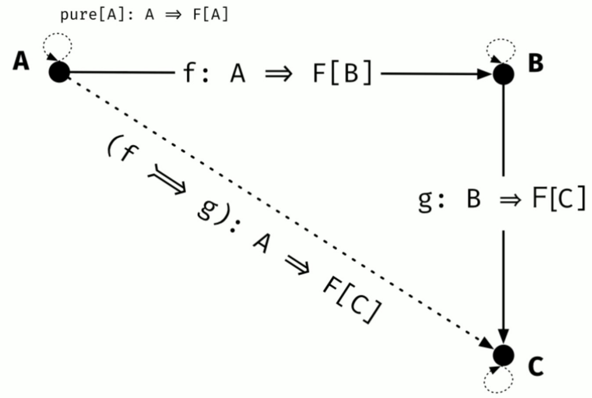
That's our problem. So what can we do?
What would it take to make them compose?

Here is our function diagram for pure function composition. And if we sort of replace things with **effectful** functions, they look like this, so we have something like **andThen**, looks something like a **fish**, and every type has an id, we are calling it **pure**. **<u>If we were able to define this and make it compose then we would get that power that we were talking about</u>.** So how do we write this in Scala?

What would it take?



What would it take?



We can implement **compose**, the **fish** operator using **flatMap**, so the **fish** operator is something we can derive later really, the operation we need is **flatMap**.

The Operations

```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]

}
```

The Operations

```scala
// 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 recap, using material by Rúnar Bjarnason, why we need Kleisli composition (the fish operator), which is defined in terms of flatMap (aka bind)

```
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**.

```
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.

Scala eXchange 2017 Keynote:
Composing Programs

Rúnar Bjarnason
@runarorama

```
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**.

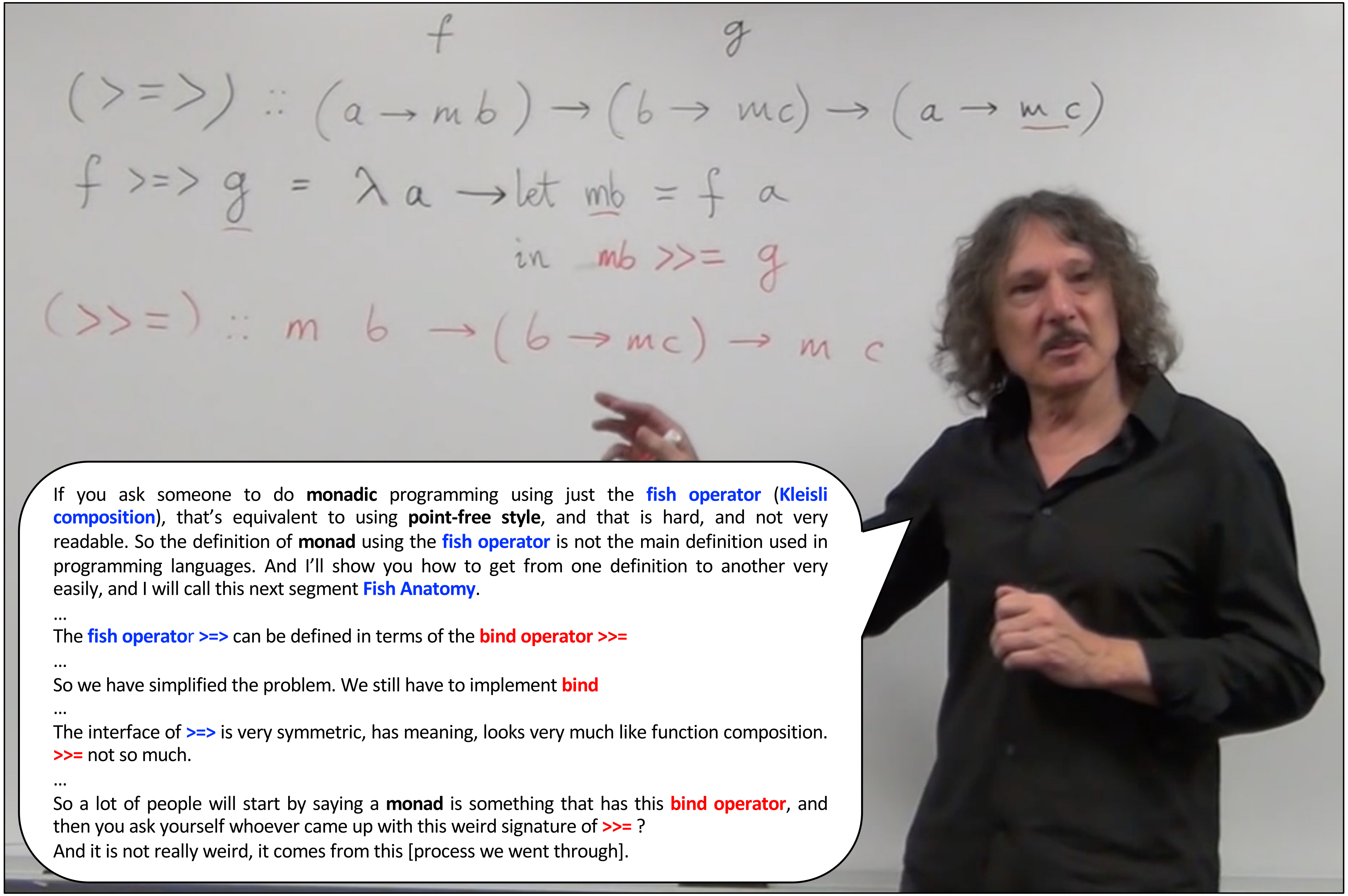
Kleisli Category

- 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
```

Bartosz Milewski defines Kleisli composition in terms of bind, in his lecture on Monads



If you ask someone to do **monadic** programming using just the **fish operator** (**Kleisli composition**), that's equivalent to using **point-free style**, and that is hard, and not very readable. So the definition of **monad** using the **fish operator** is not the main definition used in programming languages. And I'll show you how to get from one definition to another very easily, and I will call this next segment **Fish Anatomy**.

...

The **fish operator >=>** can be defined in terms of the **bind operator >>=**

...

So we have simplified the problem. We still have to implement **bind**

...

The interface of **>=>** is very symmetric, has meaning, looks very much like function composition. **>>=** 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].

Scala

Kleisli Composition (fish operator)	>=>	compose
Bind	>>=	flatMap
lifts **a** to **m a** (lifts A to F[A])	return	unit/pure

Haskell

Defining a Monad in terms of Kleisli composition and Kleisli identity function

Kleisli composition + unit

```
trait Monad[F[_]] {
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
  def unit[A](a: => A): F[A]
}
```

Kleisli composition + return

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

Defining Kleisli composition in terms of flatMap (bind)

```
def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
a => flatMap(f(a))(g)
```

```
(>=>)::(a ->mb)->(b->mb)->(a->mc)
(>=>) = \a -> (f a) >>= g
```

Defining a Monad in terms of flatmap (bind) and unit (return)

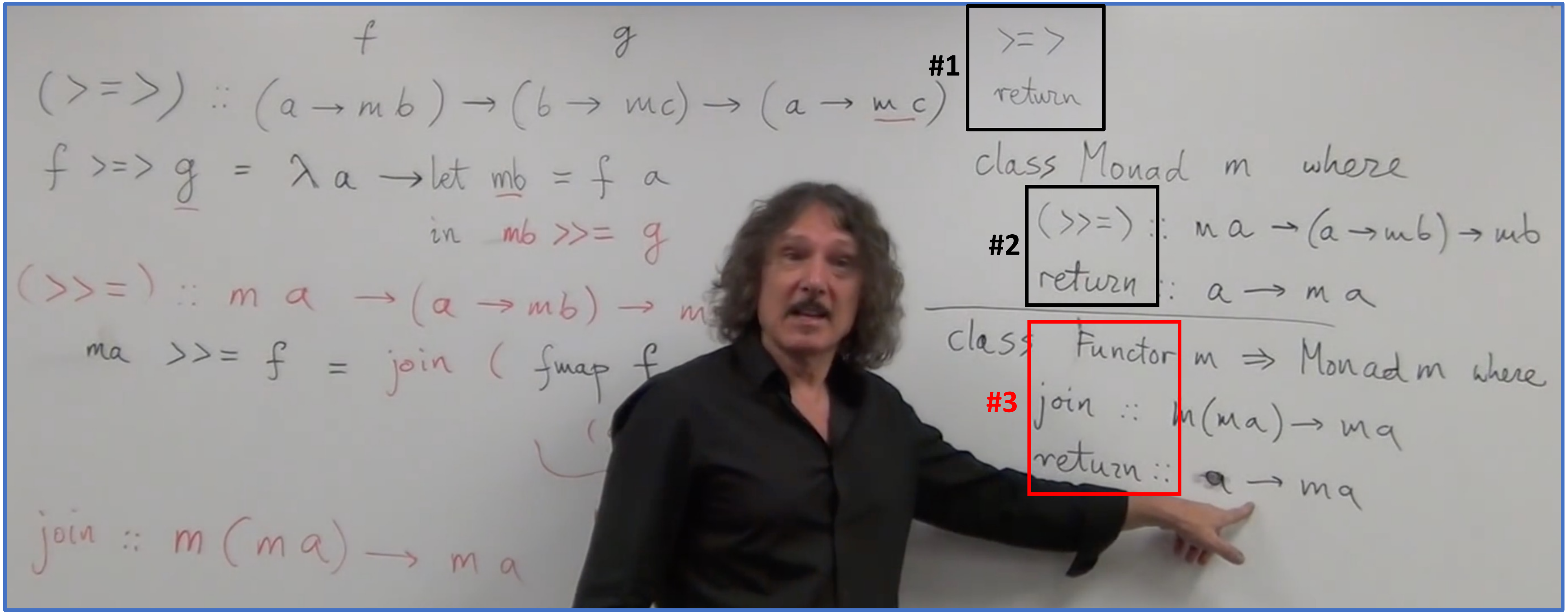
flatMap + unit

```
trait Monad[F[_]] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]

  // can then implement Kleisli composition using flatMap
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] =
    a => flatMap(f(a))(g)
}
```

bind + return (Kleisli composition can then be implemented with bind)

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

  -- can then implement Kleisli composition using bind
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  (>=>) = \a -> (f a) >>= g
```

Bartosz Milewski introduces **a third definition of Monad** in terms of join and return, based on Functor



So this (**join** and **return**) is **an alternative definition of a monad**. But 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.

…

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 **subsume functoriality**, whereas here, functoriality is separate, separately it is a functor and separately we define **join**, and separately we define **return**.

…

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 (**>>=** and **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, you don't have to do it.

In ’FP in Scala’ we also see **a third minimal sets of primitive Monad combinators**

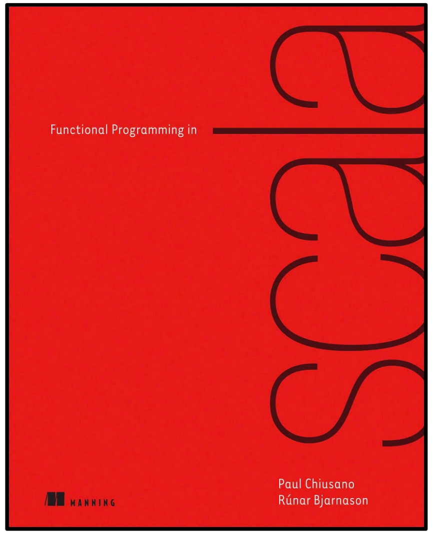
We’ve seen three minimal sets of primitive `Monad` combinators, and instances of `Monad` will have to provide implementations of one of these sets:

- `unit` and `flatMap`
- `unit` and `compose`
- `unit` , `map` , and `join`

And we know that there are two monad laws to be satisfied, associativity and identity, that can be formulated in various ways. So we can state plainly what a monad is :

> *A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.*

That’s a perfectly respectable, precise, and terse definition. And if we’re being precise, this is the only correct definition.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Scala

Haskell

flatmap + unit

```scala
trait Monad[F[_]] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]
}
```

bind + return

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

Kleisli composition + unit

```scala
trait Monad[F[_]] {
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
  def unit[A](a: => A): F[A]
}
```

Kleisli composition + return

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

Defining a Monad in terms of map (fmap), join and unit (return)

map + join + unit

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

trait Monad[F[_]] extends Functor[F] {
  def join[A](mma: F[F[A]]): F[A]
  def unit[A](a: => A): F[A]
}
```

fmap + join + return

```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
```

flatmap + unit

```
trait Monad[F[_]] {

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]

  def join[A](mma: F[F[A]]): F[A] = flatMap(mma)(ma => ma)
  def map[A,B](m: F[A])(f: A => B): F[B] = flatMap(m)(a => unit(f(a)))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = a => flatMap(f(a))(g)
}
```

defining **join**, **map** and **compose** in terms of **flatMap** and **unit**

Kleisli composition + unit

```
trait Monad[F[_]] {

  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
  def unit[A](a: => A): F[A]

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B] = compose((_:Unit) => ma, f)(())
  def map[A,B](m: F[A])(f: A => B): F[B] = flatMap(m)(a => unit(f(a)))
}
```

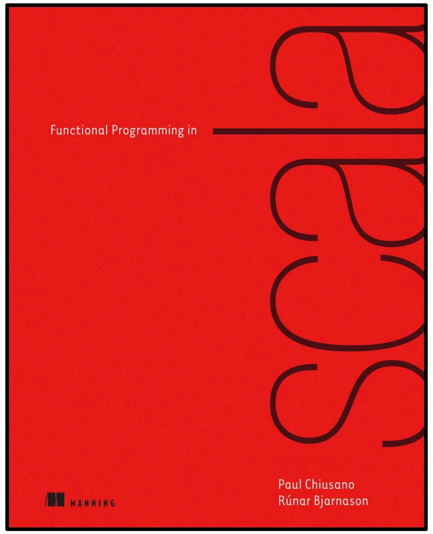
defining **flatMap** and **map** in terms of **compose** and **unit**

map + join + unit

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

trait Monad[F[_]] extends Functor[F] {

  def join[A](mma: F[F[A]]): F[A]
  def unit[A](a: => A): F[A]

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B] = join(map(ma)(f))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = a => flatMap(f(a))(g)
}
```

defining **flatMap** and **compose** in terms of **join** and **map**

Using primitive monad combinators to define the other key combinators



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

In Category Theory **a Monad is a functor equipped with a pair of natural transformations** satisfying the laws of associativity and identity

Monads in Category Theory

In Category Theory, a Monad is a functor equipped with a pair of natural transformations satisfying the laws of associativity and identity.

What does this mean? If we restrict ourselves to the category of Scala types (with Scala types as the objects and functions as the arrows), we can state this in Scala terms.

A Functor is just a type constructor for which map can be implemented:

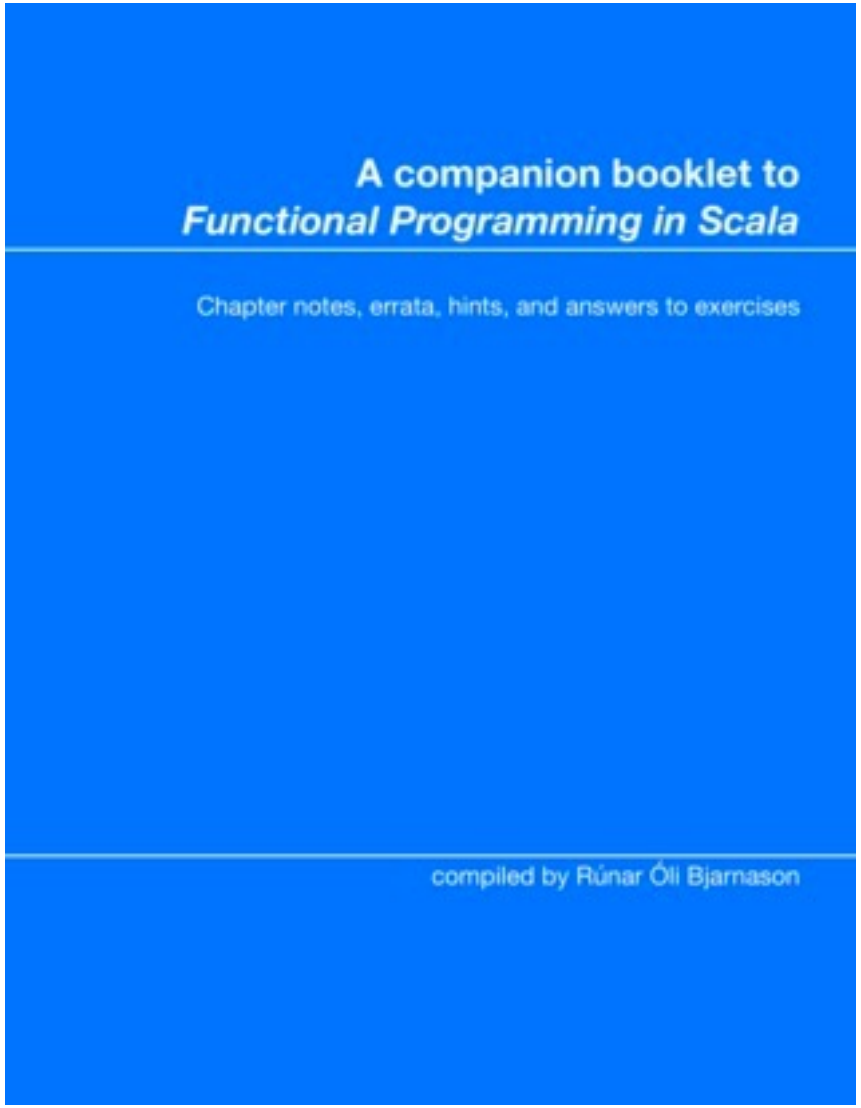
```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

A natural transformation from a functor F to a functor G is just a polymorphic function:

```
trait Transform[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

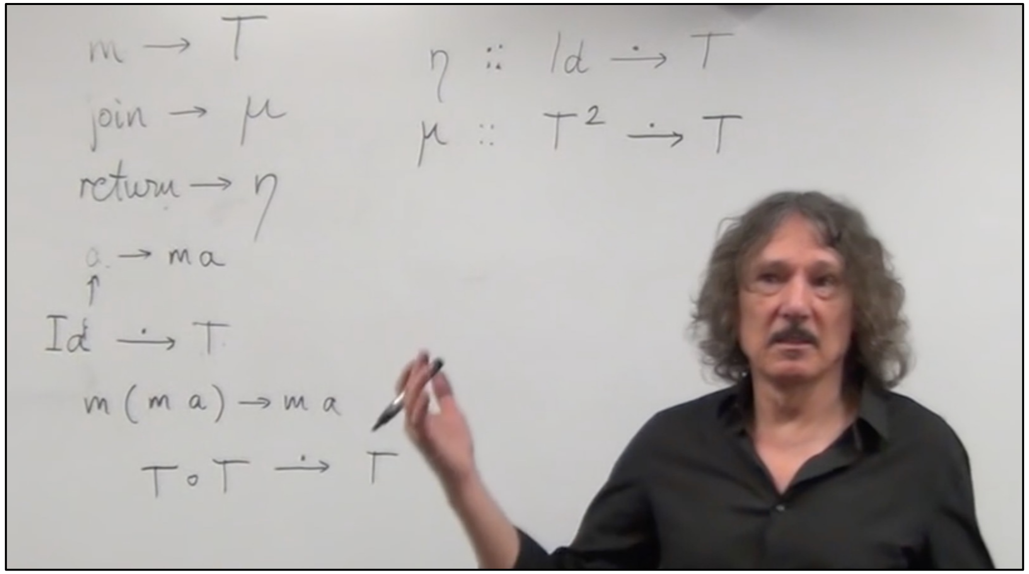
The natural transformations that form a monad for F are unit and join:

```
type Id[A] = A

def unit[F](implicit F: Monad[F]) = new Transform[Id, F] {
  def apply(a: A): F[A] = F.unit(a)
}

def join[F](implicit F: Monad[F]) = new Transform[({type f[x] = F[F[x]]})#f, F] {
  def apply(ffa: F[F[A]]): F[A] = F.join(ffa)
}
```



(by Runar Bjarnason)
@runarorama

Bartosz Milewski defines a monad as a functor and two natural transformations, plus associativity and identity laws



Category Theory 10.1: Monads

@BartoszMilewski

All three definitions [of **Monad**] are used in category theory, but really, everybody uses the one with **join**, except that they don't call it **join** and they don't call it **return**. And they don't call the **functor** **M**, they call it **T**. So the translation is **m** goes to **T**, **join** goes to **μ** and **return** goes to **η**. They use Greek letters here and it makes sesne because they use Greek letters for **natural transformations** and you will see that these (**join** and **return**) are natural transformations. So now I'll switch notation to **T**, **μ** and **η**.

We already talked about **return** at some point, when I talked about natural transformations, I said that return really is a **polymorphic function** that goes from **a** to **ma**, in the old notation, but really, since it is a polymorphic function, it is really a natural transformation **a** --> **ma** where **a** is the identity functor acting on **a** (Id_a). So it is really a natural transformation between two functors, and Id_a is a component of the natural transformation, for some particular **a**. So **return** is a component of the natural transformation from the **identity functor** to **m**, and since we don't want to use **m** here, I am going to use **T**.

So we'll say that **η** (**unit/return**) is a natural transformation from the **identity functor** to **T**. And it means the same thing, except that in **Haskell** we use it in components, every natural transformation has components, so for a particular **a**, $\mathbf{Id_a}$ acting on **a** gives you **a**, **T** acting on **a** gives you **ma**.

Now what is **μ**?, **μ** (join) is also a natural transformation. Remember, join was going from **m(m a)** to **ma**. What **is m(m a)**? It means we take this functor, we act on **a**, and then we apply it to the result. So **this is double application of the functor**. It is just **composition of the functor with itself**. So this in mathematical notation would be **T ∘ T --> T**. Double application ot **T**, in components, it will give you **m(m a)**, double application of a functor in components is **m(m a)**. Single application of the functor is **m a**. It is a natural transformation.

So in category theory we have to say it is a natural transformation. In **Haskell** we didn't say it is natural transformation, we didn't mention the **naturality condition** for **return**. Why? Because of **polymorphism**, because of 'theorems for free', it is automatically a natural transformation, the naturality condition is automatic. But in category theory we have to say it is a natural transformation. So **T ∘ T** is usually written simply as $\mathbf{T^2}$. $\mathbf{T^2}$ is just compositon of **T** with itself. That's shorthand notation.

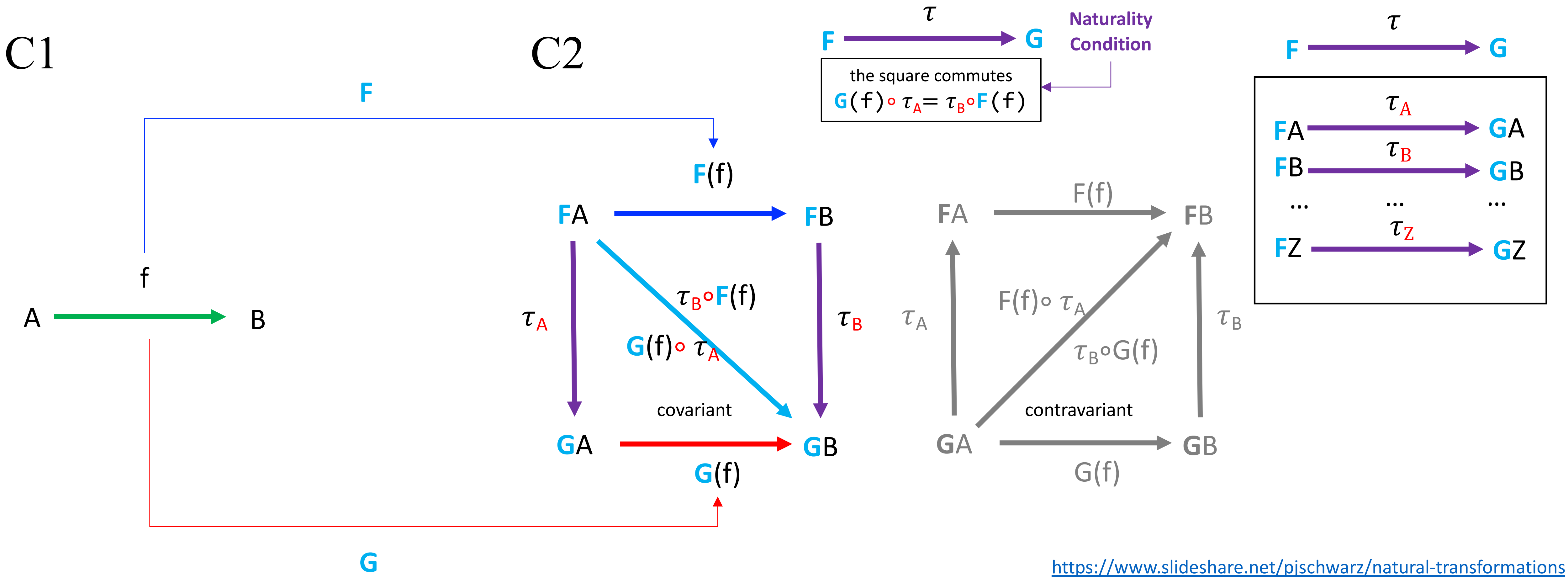
So, a monad is a functor T and two natural transformations. Plus some laws, otherwise if we try to make a Kleisli category on top of this we wouldn't get **associativity and identity laws**.

Let’s recap on Natural Transformations (3 slides)

Natural Transformation

C1 and C2 are categories and ∘ denotes their composition operations.
F and G are functors from C1 to C2 which map each C1 object to a C2 object and map each C1 arrow to a C2 arrow
A natural transformation τ from F to G (either both covariant of both contravariant) is
a family of arrows τ_X: FX → GX of C2 indexed by the object X of C1 such that for each arrow f: A → B of C1, the appropriate square in C2 commutes (depending on the variance)

natural transformation
τ from F to G



https://www.slideshare.net/pjschwarz/natural-transformations

Generic Scala Example: Natural Transformation between two Functors from the category of 'Scala types and functions' to itself

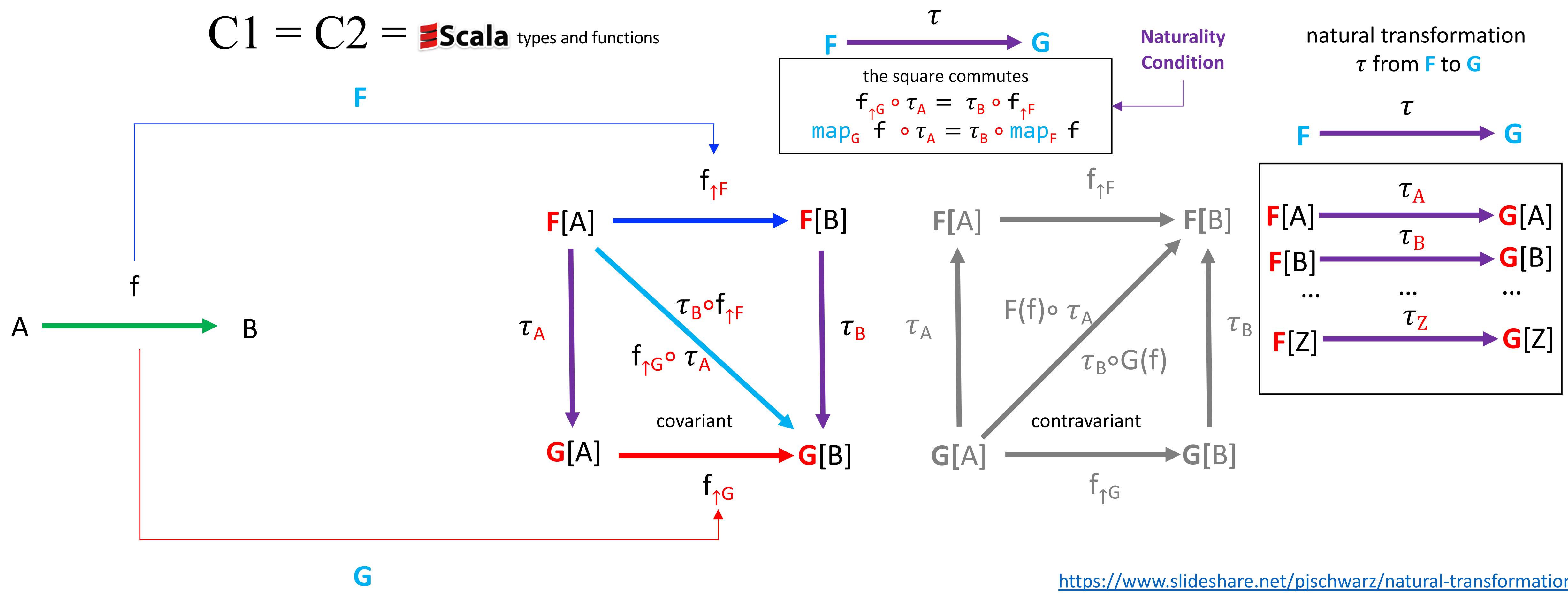
C1 = C2 = Scala types and functions
- **objects**: types
- **arrows**: functions
- **composition operation**: compose function, denoted here by ∘
- **identity arrows**: identity function T => T

Functor F from C1 to C2 consisting of
- type constructor F that maps type A to F[A]
- a map function from function f:A=>B to function $f_{\uparrow F}$:F[A] => F[B]

Functor G from C1 to C2 consisting of
- type constructor G that maps type A to G[A]
- a map function from function f:A=>B to function $f_{\uparrow G}$:G[A] => G[B]

F[A] is type A lifted into context F
$f_{\uparrow F}$ is function f lifted into context F

map lifts f into F
$f_{\uparrow F}$ is map f



https://www.slideshare.net/pjschwarz/natural-transformations

Concrete Scala Example: safeHead - natural transformation τ from List functor to Option functor

```scala
val length: String => Int = s => s.length

// a natural transformation
def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}
```

natural transformation τ from List to Option

$$\text{List} \xrightarrow{\tau} \text{Option}$$

the square commutes

safeHead ∘ $\text{length}_{\uparrow\text{List}}$ = $\text{length}_{\uparrow\text{Option}}$ ∘ safeHead

safeHead ∘ (map_{List} length) = ($\text{map}_{\text{Option}}$ length) ∘ safeHead

Naturality Condition

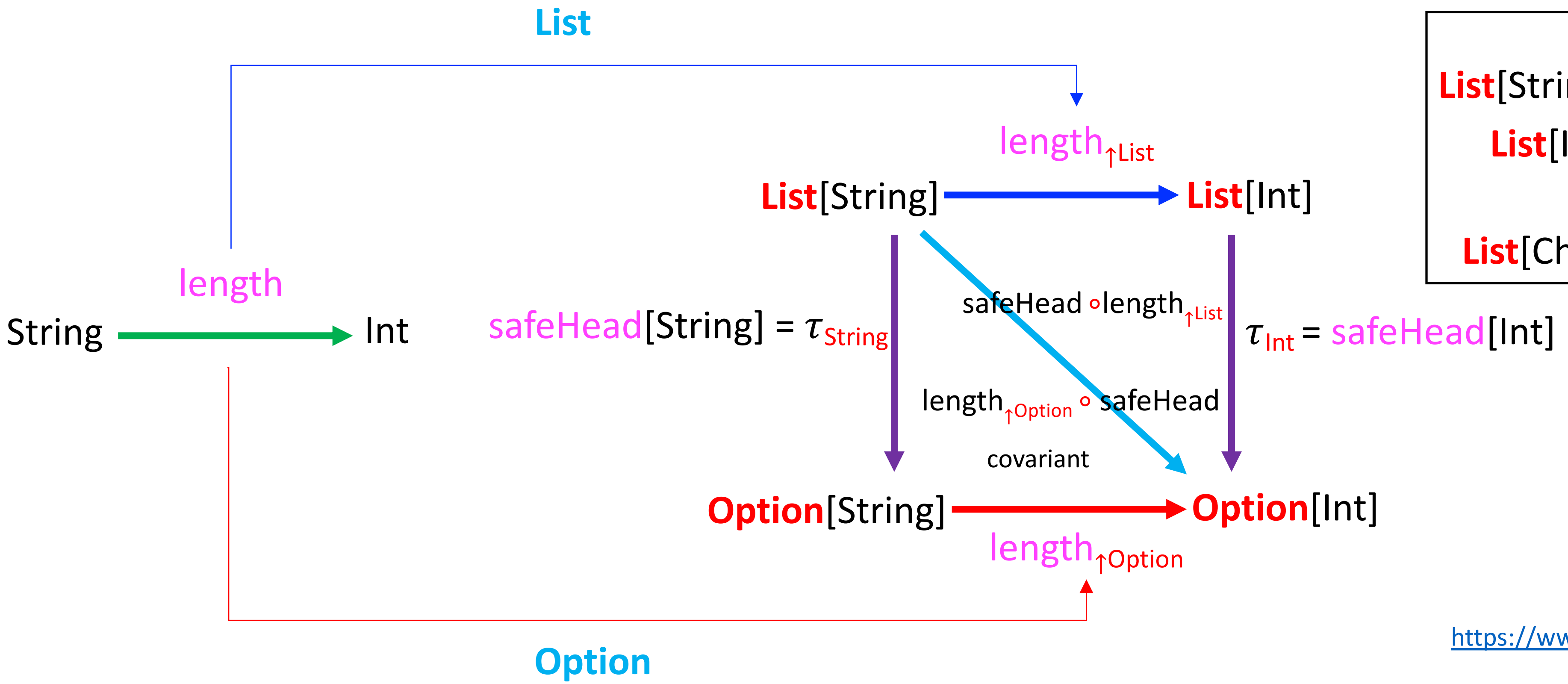
F[A] is type A lifted into context F
$f_{\uparrow F}$ is function f lifted into context F

map lifts f into F
$f_{\uparrow F}$ is map f

C1 = C2 = Scala types and functions

$$\text{List} \xrightarrow{\tau} \text{Option}$$

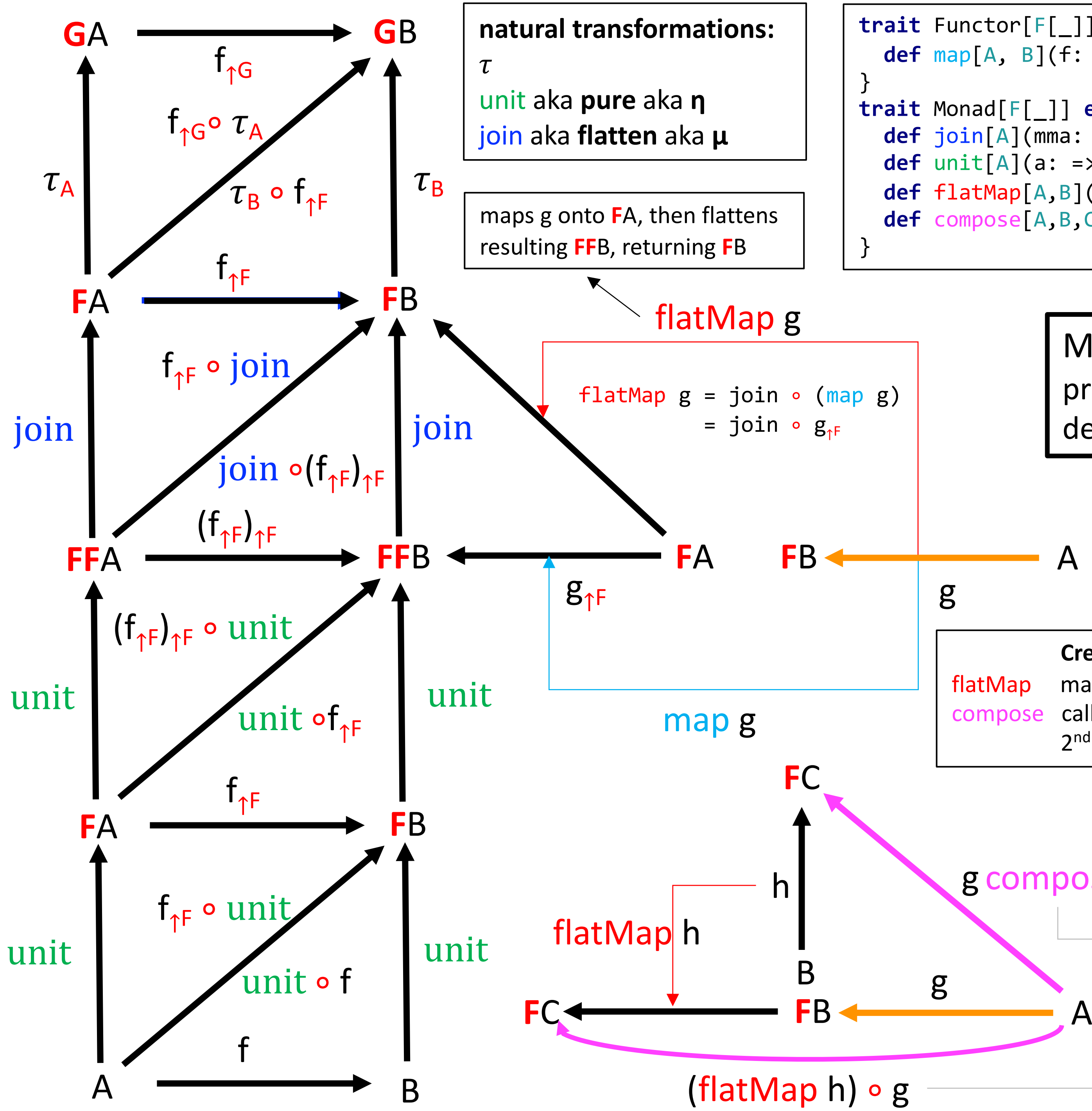
$$\text{List[String]} \xrightarrow{\tau_{\text{String}}} \text{Option[String]}$$
$$\text{List[Int]} \xrightarrow{\tau_{\text{Int}}} \text{Option[Int]}$$
$$\ldots \quad \ldots \quad \ldots$$
$$\text{List[Char]} \xrightarrow{\tau_{\text{Char}}} \text{Option[Char]}$$



Monadic Combinators in Action

primitive: unit and join – natural transformations

derived: flatMap and compose (Kleisli composition)




```scala
trait Functor[F[_]] {
  def map[A, B](f: A => B): F[A] => F[B]
}
trait Monad[F[_]] extends Functor[F] {
  def join[A](mma: F[F[A]]): F[A]
  def unit[A](a: => A): F[A]
  def flatMap[A,B](f: A => F[B]): F[A] => F[B] = (ma:F[A]) => join(map(f)(ma))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = a => flatMap(g)(f(a))
}
```

Note: signatures of map and flatMap have here been rearranged (by swapping parameters around and uncurrying): they now return a function

f	**pure** function
g,h	**effectful** (**kleisli**) functions

	Creates a function which
flatMap	maps given function over its argument and joins the result
compose	calls 1st given function with its argument and flatMaps 2nd given function over the result

calls g with A, then flatMaps h over result FB (to apply h to its contents), returning result FC.

g compose h = (flatMap h) ∘ g

flatMap	>>=
compose	>=>
map	<$>

map lifts f into F

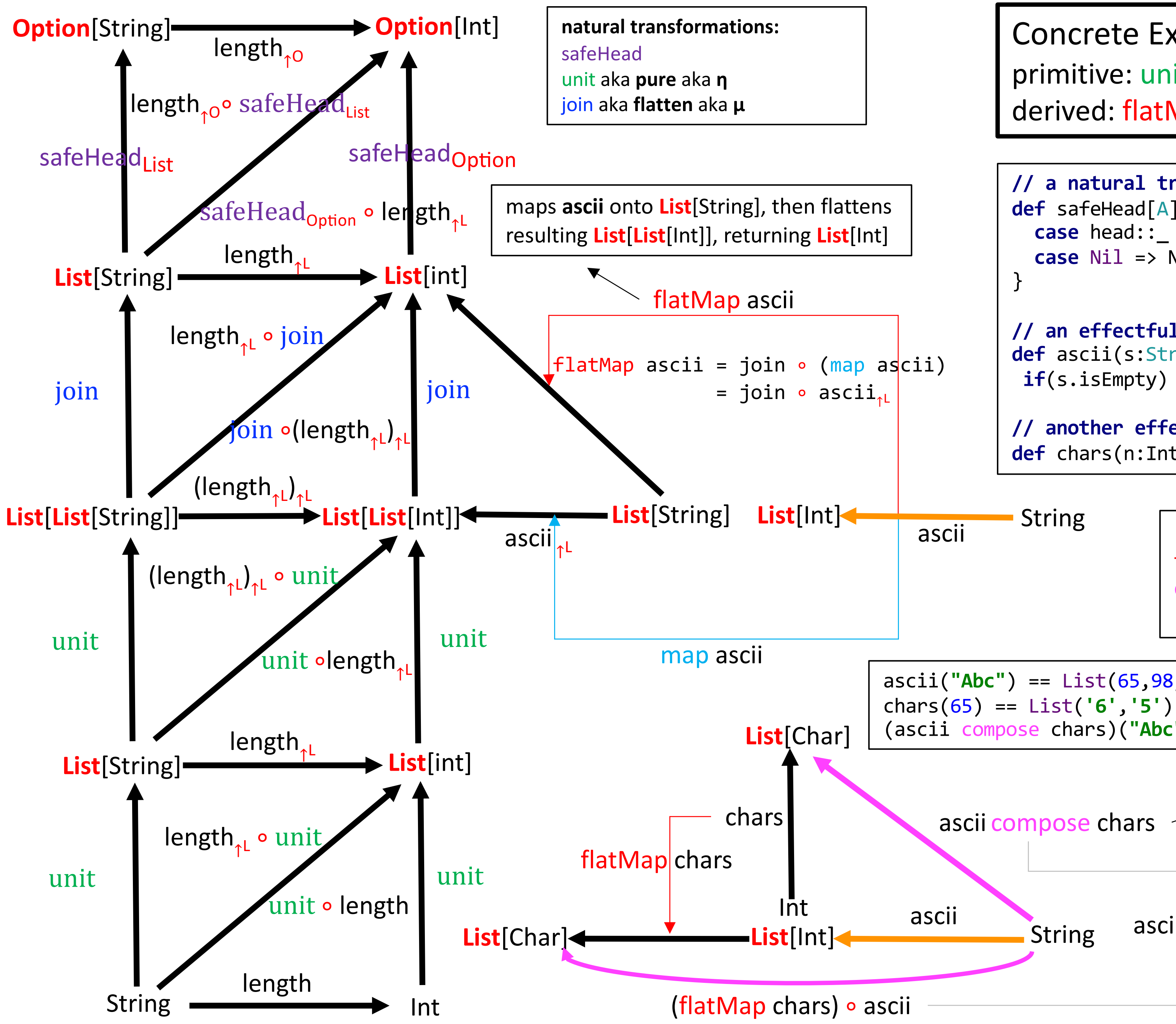
$f_{\uparrow F}$ is map f

Concrete Example of Monadic Combinators in Action

primitive: unit and join – natural transformations

derived: flatMap and compose (Kleisli composition)

```scala
// a natural transformation
def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}

// a pure function
val length: String => Int =
  s => s.length

// an effectful (Kleisli) function - ascii("Abc") == List(65,98,99)
def ascii(s:String): List[Int] =
 if(s.isEmpty) Nil else s.head.toInt :: ascii(s.tail)

// another effectful function - chars(65) == List('6','5')
def chars(n:Int): List[Char] = n.toString.toList
```



	Creates a function which
flatMap	maps given function over its argument and joins the result
compose	calls 1st given function with its argument and flatMaps 2nd given function over the result

```scala
ascii("Abc") == List(65,98,99)
chars(65) == List('6','5')
(ascii compose chars)("Abc") = List('6','5','9','8','9','9')
```

map lifts f into **F**

$f_{\uparrow L}$ is map f for **F=List**

$f_{\uparrow O}$ is map f for **F=Option**

calls **ascii** with a String, then flatMaps **chars** onto result **List**[Int] (producing a Char for each Int), returning result **List**[Char]

ascii compose chars = (flatMap chars) ∘ ascii

```scala
trait Functor[F[_]] {
  def map[A, B](f: A => B): F[A] => F[B]
}
trait Monad[F[_]] extends Functor[F] {
  def join[A](mma: F[F[A]]): F[A]
  def unit[A](a: => A): F[A]

  def flatMap[A,B](f: A => F[B]): F[A] => F[B] =
    (ma:F[A]) => join(map(f)(ma))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] =
    a => flatMap(g)(f(a))
}
```

```scala
def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}
def ascii(s:String):List[Int] =
  if(s.isEmpty) Nil else s.head.toInt :: ascii(s.tail)

def chars(n:Int):List[Char] =
  n.toString.toList

assert(ascii("Abc") == List(65,98,99))
assert(chars(65) == List('6','5'))
```

```scala
val listM = new Monad[List] {
  def map[A, B](f: A => B): List[A] => List[B] = {
    case head :: tail => f(head) :: map(f)(tail)
    case Nil => Nil
  }
  def unit[A](a: => A): List[A] = List(a)
  def join[A](mma: List[List[A]]): List[A] =
    mma match {
      case head::tail => head ::: join(tail)
      case Nil => Nil
  }
}
```

```scala
val optionM = new Monad[Option] {
  def map[A,B](f: A => B): Option[A] => Option[B] = {
    case Some(a) => Some(f(a))
    case None => None
  }
  def unit[A](a: => A): Option[A] = Some(a)
  def join[A](mma: Option[Option[A]]): Option[A] =
    mma match {
      case Some(a) => a
      case None => None
  }
}
```

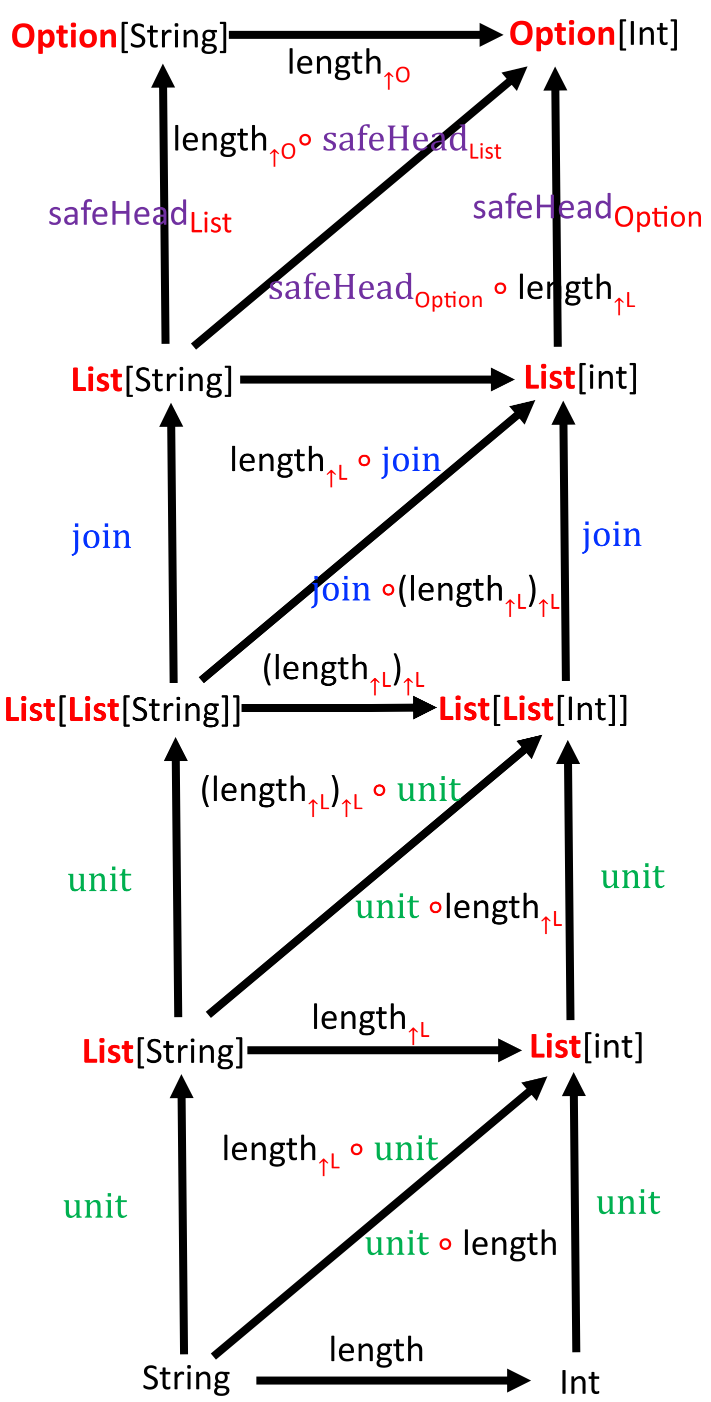
```scala
val length: String => Int = s => s.length
val lengthLiftedOnce: List[String] => List[Int] = (listM map length)
val lengthLiftedTwice: List[List[String]] => List[List[Int]] = listM map lengthLiftedOnce
assert(length("abcd") == 4)
assert(lengthLiftedOnce(List("abcd","efg","hi")) == List(4,3,2))
assert(lengthLiftedTwice(List(List("abcd","efg","hi"),List("jkl","mo","p"))) == List(List(4,3,2),List(3,2,1)))
```

```scala
val computeAndThenTransform: List[String] => Option[Int] = safeHead compose (listM map length)
val transformAndThenCompute: List[String] => Option[Int] = (optionM map length) compose safeHead

assert(computeAndThenTransform(List("abc", "d", "ef")) == transformAndThenCompute(List("abc", "d", "ef")))
assert(computeAndThenTransform(List("abc", "d", "ef")) == Some(3))
assert(transformAndThenCompute(List("abc", "d", "ef")) == Some(3))

assert(computeAndThenTransform(List()) == transformAndThenCompute(List()))
assert(computeAndThenTransform(List()) == None)
assert(transformAndThenCompute(List()) == None)
```

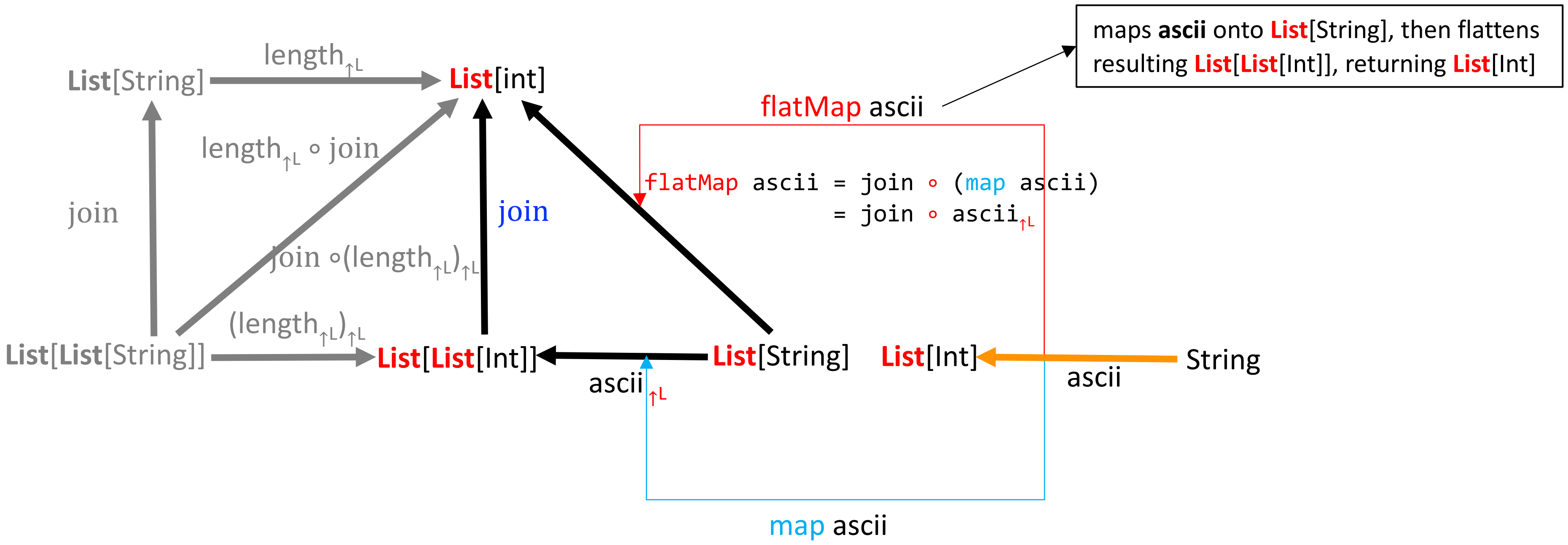
```scala
val computeAndThenTransform: List[List[String]] => List[Int]  = (listM.join[Int](_)) compose lengthLiftedTwice
val transformAndThenCompute: List[List[String]] => List[Int]  = lengthLiftedOnce compose listM.join

assert(computeAndThenTransform(List(List("abcd","efg","hi"),List("jkl","mo","p"))) ==
  transformAndThenCompute(List(List("abcd","efg","hi"),List("jkl","mo","p"))))
assert(computeAndThenTransform(List(List("abcd","efg","hi"),List("jkl","mo","p"))) == List(4,3,2,3,2,1))
assert(transformAndThenCompute(List(List("abcd","efg","hi"),List("jkl","mo","p"))) == List(4,3,2,3,2,1))
```

```scala
val computeAndThenTransform: List[String] => List[List[Int]]  =
  ((x:List[Int]) => listM.unit(x)) compose lengthLiftedOnce
val transformAndThenCompute: List[String] => List[List[Int]]  =
  lengthLiftedTwice compose ((x:List[String]) => listM.unit(x))

assert(computeAndThenTransform(List("abcd","efg","hi")) == transformAndThenCompute(List("abcd","efg","hi")))
assert(computeAndThenTransform(List("abcd","efg","hi")) == List(List(4,3,2)))
assert(transformAndThenCompute(List("abcd","efg","hi")) == List(List(4,3,2)))
```

```scala
val computeAndThenTransform: String => List[Int]  = ((x:Int) => listM.unit(x)) compose length
val transformAndThenCompute: String => List[Int]  = lengthLiftedOnce compose ((x:String) => listM.unit(x))

assert(computeAndThenTransform("abcd") == transformAndThenCompute("abcd"))
assert(computeAndThenTransform("abcd") == List(4))
assert(transformAndThenCompute("abcd") == List(4))
```



```scala
def ascii(s:String):List[Int] = if(s.isEmpty) Nil else s.head.toInt :: ascii(s.tail)
assert(ascii("Abc") == List(65,98,99))

val mappedAscii: List[String] => List[List[Int]] = listM map ascii
val flatMappedAscii: List[String] => List[Int] = listM flatMap ascii
val compositionOfMappedAsciiAndJoin = (listM.join[Int](_)) compose mappedAscii

assert( compositionOfMappedAsciiAndJoin(List("abcd","efg","hi")) == flatMappedAscii(List("abcd","efg","hi")) )
assert( compositionOfMappedAsciiAndJoin(List("abcd","efg","hi")) == List(97, 98, 99, 100,101,102,103,104,105))
assert( flatMappedAscii(List("abcd","efg","hi")) == List(97, 98, 99, 100,101,102,103,104,105))
```



```scala
def ascii(s:String):List[Int] = if(s.isEmpty) Nil else s.head.toInt :: ascii(s.tail)
def chars(n:Int):List[Char] = n.toString.toList

assert(chars(65) == List('6','5'))
assert(ascii("Abc") == List(65,98,99))

def flatMappedChars = listM flatMap chars
def kleisliCompositionOfAsciiAndChars = listM.compose(ascii, chars)

assert( kleisliCompositionOfAsciiAndChars("Abc") == (flatMappedChars compose ascii)("Abc"))
assert( kleisliCompositionOfAsciiAndChars("Abc") == List('6','5','9','8','9','9'))
assert( (flatMappedChars compose ascii)("Abc") == List('6','5','9','8','9','9'))
```