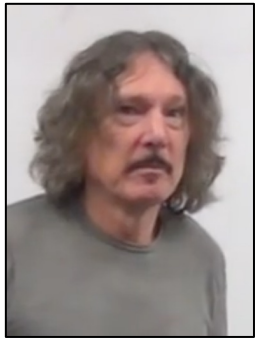
Functor Composition

including (starting from) the definition in



@BartoszMilewski

slides by  @philip_schwarz

Let's look at **Bartosz Milewski**'s explanation of the fact that **Functors compose**.

If you need an introduction to **Functors** then see the following

@philip_schwarz

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/functors

https://www.slideshare.net/pjschwarz/functor-laws

7.1 Functors in Programming

```
def map[A, B](f: A => B)(fa: Option[A]): Option[B]
```

7.1.1 The Option Functor

```
def map[A, B](f: A => B): Option[A] => Option[B] = {
  case None => None
  case Some(x) => Some(f(x))
}
```

7.1.4 Typeclasses

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

implicit val optionFunctor = new Functor[Option] {
  def map[A, B](f: A => B)(fa: Option[A]): Option[B] =
    fa match {
      case None => None
      case Some(x) => Some(f(x))
    }
  }
```

7.1.6 The List Functor

```
implicit val listFunctor = new Functor[List] {
  def map[A, B](f: A => B)(fa: List[A]): List[B] =
    fa match {
      case Nil => Nil
      case x :: t => f(x) :: map(f)(t)
    }
  }
```

7.3 Functor Composition

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

```
def maybeTail[A]: List[A] => Option[List[A]] = {
  case Nil => None
  case x :: xs => Some(xs)
}
```

Scala Edition Contains code snippets in **Haskell** and **Scala**

The result of maybeTail is of a type that's a composition of two functors, Option and **List**, acting on **A**. Each of these functors is equipped with its own version of **map**, but what if we want to apply some function **f** to the contents of the **composite**: an **Option** of a **List**? We have to break through two layers of functors. We can use **map** to break through the outer **Option**. But we can't just send **f** inside **Option** because **f** doesn't work on lists. We have to send (**map f**) to operate on the inner list. For instance, let's see how we can **square** the elements of an **Option** of a **List** of integers:

```
def square: Int => Int = x => x * x
val maybeList: Option[List[Int]] = Some(1 :: (2 :: (3 :: Nil)))
val maybeListSquared = (optionFunctor map (listFunctor map square))(maybeList)
assert( maybeListSquared == Some(1 :: (4 :: (9 :: Nil))) )
```

The compiler, after analyzing the types, will figure out that, for the outer map, it should use the implementation from the **Option** instance, and for the inner one, the **List functor** implementation.

```scala
def square: Int => Int = x => x * x
val maybeList: Option[List[Int]] = Some(1 :: (2 :: (3 :: Nil)))
val maybeListSquared = (optionFunctor map (listFunctor map square))(maybeList)
```

It may not be immediately obvious that the last line in the above code may be rewritten as:

```scala
def mapOption[A, B]: (A => B) => Option[A] => Option[B] = optionFunctor.map

def mapList[A, B]: (A => B) => List[A] => List[B] = listFunctor.map

def mapOptionList[A, B]: (A => B) => Option[List[A]] => Option[List[B]] =
  mapOption compose mapList

val maybeListSquared = mapOptionList(square)(maybeList)
```

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

```scala
def map[F[_], A, B]: (A => B) => (F[A] => F[B])
```

In our case, the second **map** in (**map**Option compose **map**List) takes as its argument:

```scala
  def square: Int => Int
```

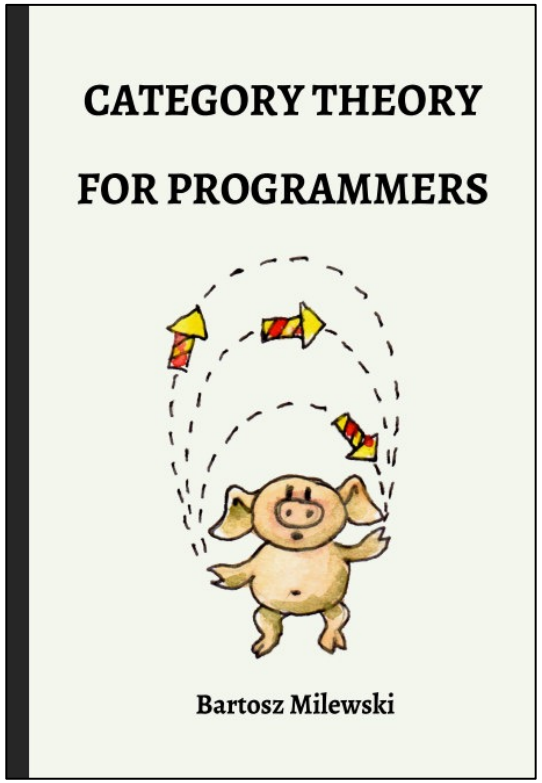
and returns a function of the type:

```scala
  List[Int] => List[Int]
```

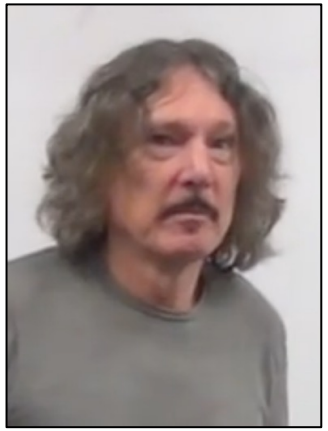
The first **map** then takes that function and returns a function:

```scala
  Option[List[Int]] => Option[List[Int]]
```

Finally, that function is applied to maybeList. So **the composition of two functors is a functor whose map is the composition of the corresponding maps**.



Scala Edition Contains code snippets in **Haskell** and **Scala**



@BartoszMilewski

Yes, to aid comprehension in our context, I have taken the liberty to rename a few things, e.g. **fmap** → **map**, **Maybe** → **Option**, a → A, **Haskell** → **Scala**, **mis** → **maybeList**, **mis2** → **maybeListSquared**, **Cons** → ::

In the next two slides I have a go at visualizing this notion of a **composite Functor**.

@philip_schwarz

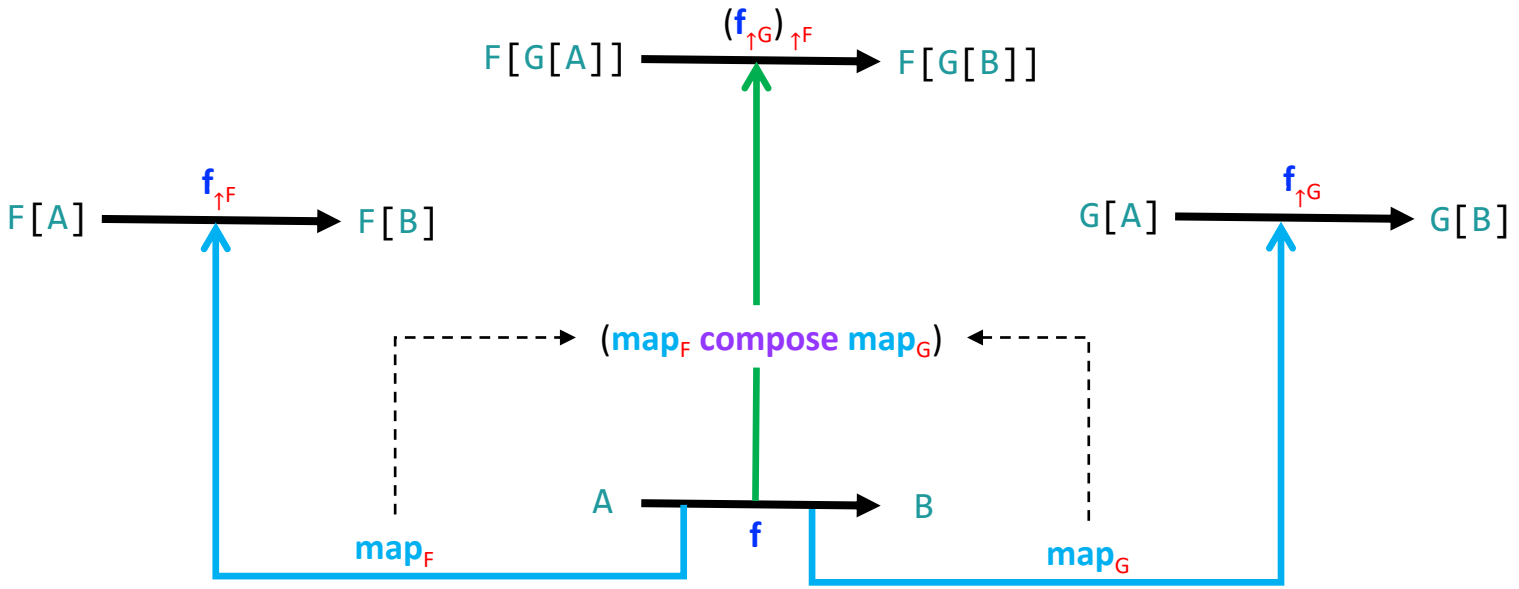
```scala
def square: Int => Int = x => x * x

def mapOption[A,B]:
  (A => B) => Option[A] => Option[B] =
  optionFunctor.map

def mapList[A,B]:
  (A => B) => List[A] => List[B] =
  listFunctor.map

def mapOptionList[A,B]:
  (A => B) => Option[List[A]] => Option[List[B]] =
  mapOption compose mapList
```

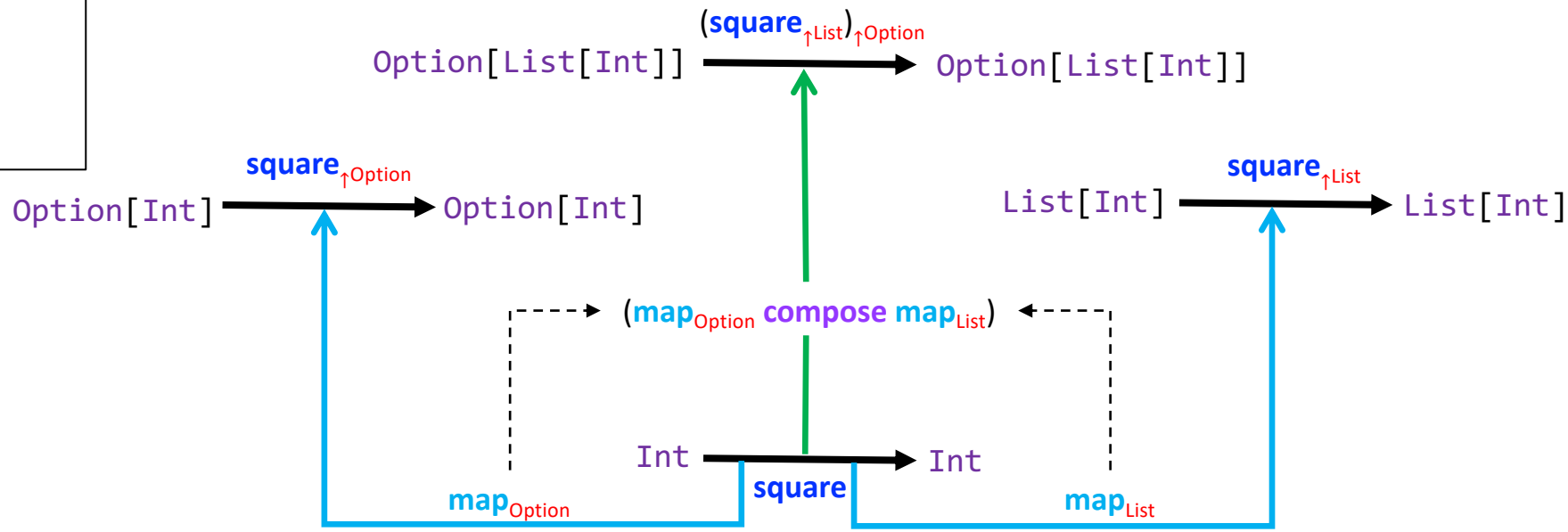
```scala
package scala
trait Function1 …
/** Composes two instances of Function1 in a new
 *  Function1, with this function applied last.
 *  @tparam   A   the type to which function
 *                `g` can be applied
 *  @param    g   a function A => T1
 *  @return       a new function `f` such that
 *                `f(x) == apply(g(x))`
 */
…def compose[A](g: A => T1): A => R =
   { x => apply(g(x)) }
…
```

map_F lifts function **f** into **F**
$f_{\uparrow F}$ is **f** lifted into **F**

map_{Option} = **mapOption**
map_{List} = **mapList**
(map_{Option} compose map_{List}) = **mapOptionList**

the **composition** of two **functors** is a **functor** whose **map** is the **composition** of the corresponding **maps**



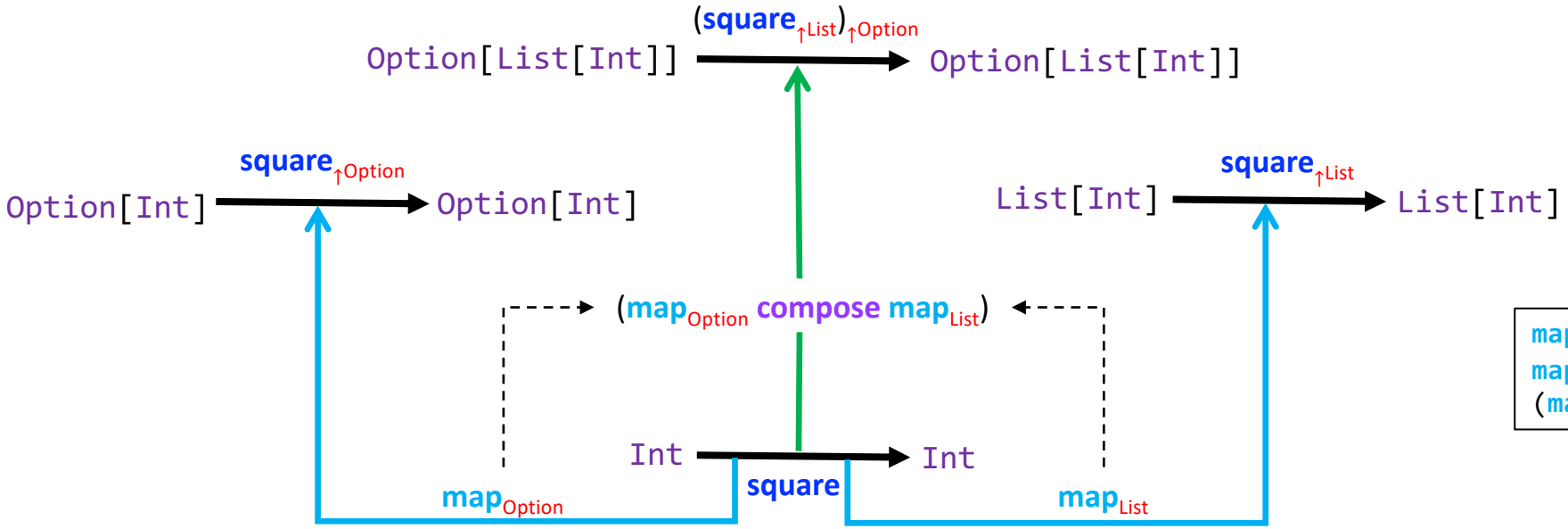


The composition of two functors is a functor whose map is the composition of the corresponding maps

```
// mapOptionList = mapOption compose mapList

assert(mapOptionList(square)(Some(List(1,2,3))) == mapOption(mapList(square))(Some(List(1,2,3))))

assert(mapOptionList(square)(Some(List(1,2,3))) == Some(List(1,4,9)))

assert((mapOption(mapList(square))(Some(List(1,2,3))) == Some(List(1,4,9)))
```

mapping f with the **composition** of two **functors** is the same as first **mapping f** with the 1st **functor** and then **mapping** the result with the 2nd **functor**.

@philip_schwarz



mapF lifts function f into F
f↑F is f lifted into F

```
mapOption = mapOption
mapList = mapList
(mapOption compose mapList) = mapOptionList
```

```
def square: Int => Int = x => x * x

def mapOption[A,B]: (A => B) => Option[A] => Option[B] = optionFunctor.map

def mapList[A,B]: (A => B) => List[A] => List[B] = listFunctor.map

def mapOptionList[A,B]: (A => B) => Option[List[A]] => Option[List[B]] = mapOption compose mapList
```

To get a fuller picture of the notion of a **composite Functor**, let's look at **Functor** laws.

See the following for a more comprehensive introduction to **Functor** laws.

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/functor-laws

A Functor 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, denoted here by $\mathbf{id_T}$

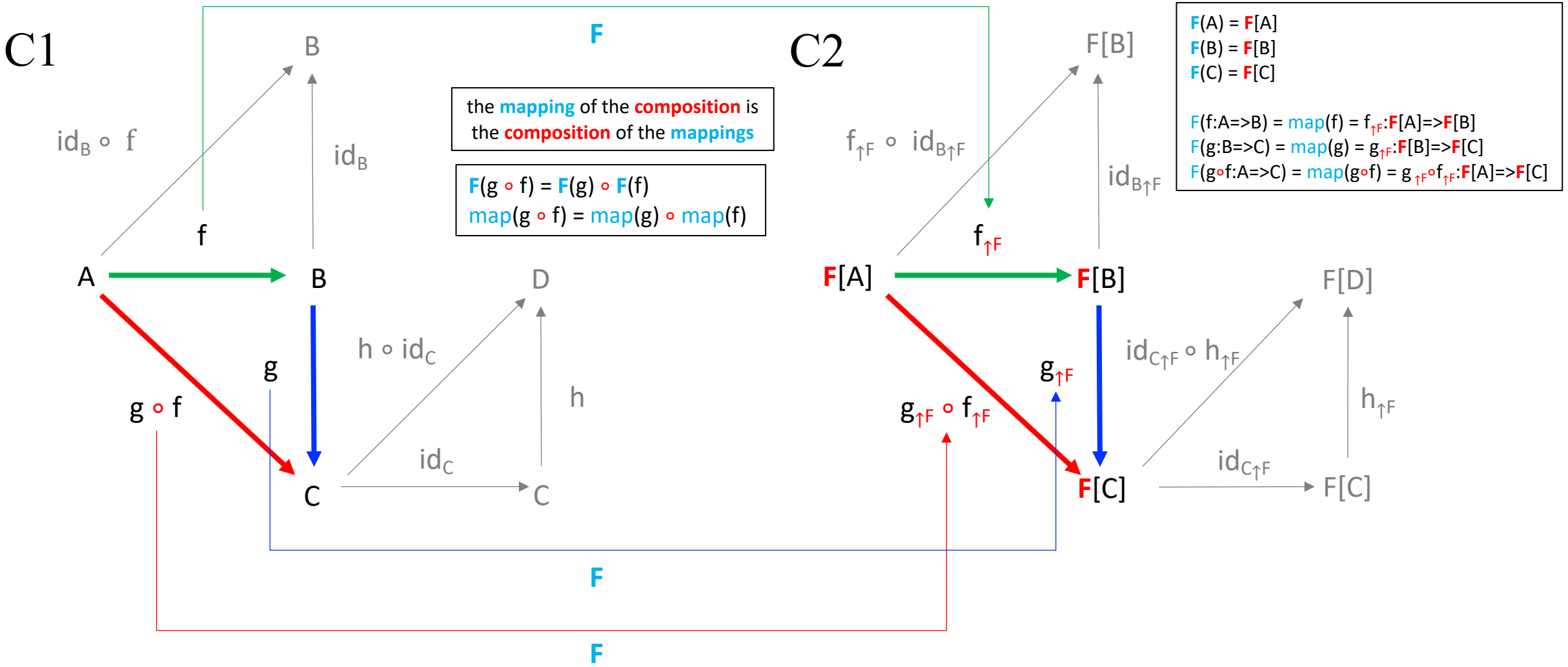
A functor F from C1 to C2 consisting of
- a type constructor **F** that maps type A to **F**[A]
- a map function from function f:A=>B to function $f_{\uparrow F}$:**F**[A] => **F**[B]

Functor Laws

$F(g \circ f) = F(g) \circ F(f)$ **i.e.** $map(g \circ f) = map(g) \circ map(f)$

$F(id_X) = id_{F(X)}$ **i.e.** $map(id_X) = id_{X\uparrow F}$

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

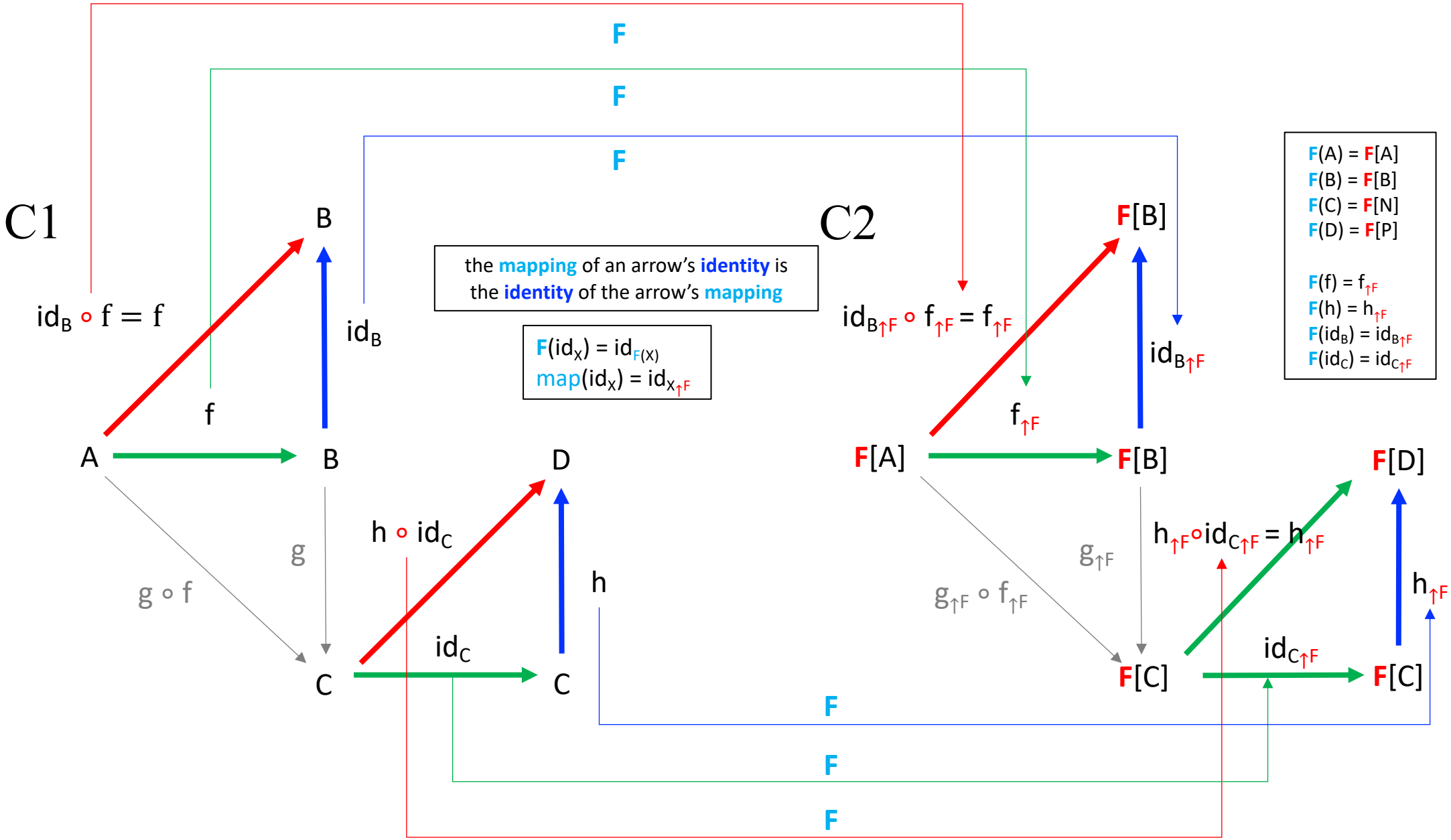


the **mapping** of the **composition** is the **composition** of the **mappings**

$F(g \circ f) = F(g) \circ F(f)$
$map(g \circ f) = map(g) \circ map(f)$

$F(A) = F[A]$
$F(B) = F[B]$
$F(C) = F[C]$

$F(f:A=>B) = map(f) = f_{\uparrow F}:F[A]=>F[B]$
$F(g:B=>C) = map(g) = g_{\uparrow F}:F[B]=>F[C]$
$F(g \circ f:A=>C) = map(g \circ f) = g_{\uparrow F} \circ f_{\uparrow F}:F[A]=>F[C]$


F
F
F
C1
C2
B
F[B]
$id_B \circ f = f$
id_B
$id_{B_{\uparrow F}} \circ f_{\uparrow F} = f_{\uparrow F}$
$id_{B_{\uparrow F}}$
the mapping of an arrow's identity is
the identity of the arrow's mapping
$F(id_X) = id_{F(X)}$
$map(id_X) = id_{X_{\uparrow F}}$
F(A) = F[A]
F(B) = F[B]
F(C) = F[N]
F(D) = F[P]
$F(f) = f_{\uparrow F}$
$F(h) = h_{\uparrow F}$
$F(id_B) = id_{B_{\uparrow F}}$
$F(id_C) = id_{C_{\uparrow F}}$
f
$f_{\uparrow F}$
A
B
D
F[A]
F[B]
F[D]
g
h ∘ id_C
h
$g_{\uparrow F}$
$h_{\uparrow F} \circ id_{C_{\uparrow F}} = h_{\uparrow F}$
$h_{\uparrow F}$
g ∘ f
$g_{\uparrow F} \circ f_{\uparrow F}$
id_C
$id_{C_{\uparrow F}}$
C
C
F[C]
F[C]
F
F
F

Now let's revisit that with the notion of a **composite Functor** in mind.

@philip_schwarz

The composition of Functors G and F is a composite functor G ∘ F from C1 to C2 consisting of

- ~~a type constructor~~
- a $map_{G \circ F}$ function from function f:A=>B to function $(f_{\uparrow F})_{\uparrow G}$: G[F[A]] => G[F[B]]

where $map_{G \circ F} = map_G \circ map_F$

Here is what the **Functor Laws** look like for a **composite Functor**

the mapping of the composition is the composition of the mappings

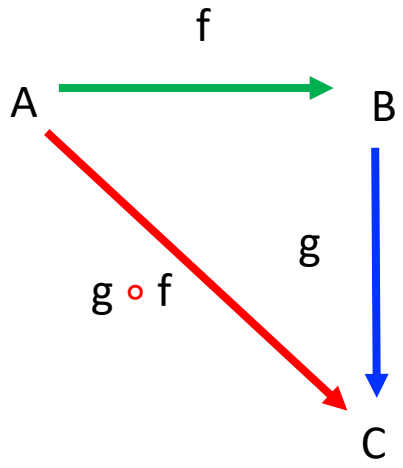
$G \circ F(g \circ f) = (G \circ F(g)) \circ (G \circ F(f))$
i.e. $map_{G \circ F}(g \circ f) = (map_{G \circ F}(g)) \circ (map_{G \circ F}(f))$

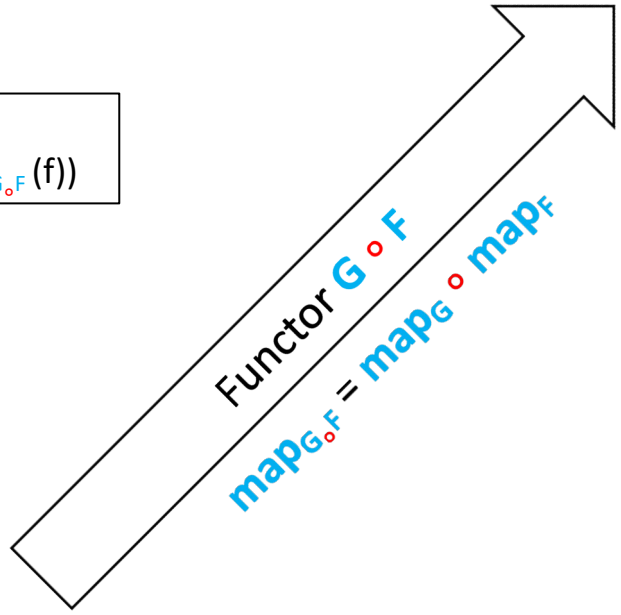
the mapping of an arrow's identity is the identity of the arrow's mapping

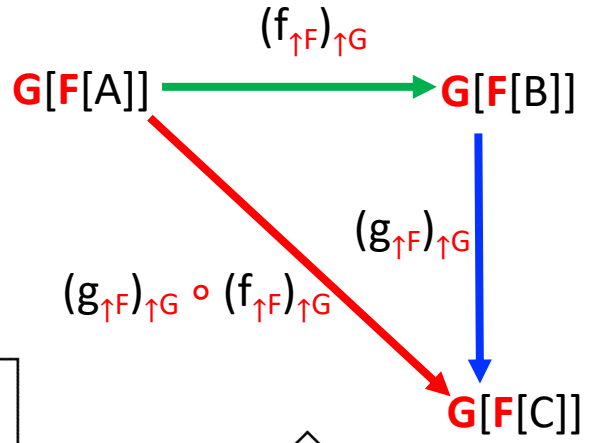
$G \circ F(id_X) = id_{G \circ F(X)}$
i.e. $map_{G \circ F}(id_X) = (id_{X \uparrow F})_{\uparrow G}$

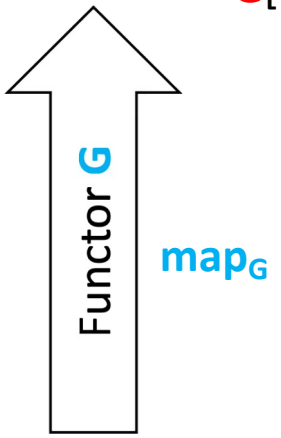
To keep the diagram simple, we are only illustrating the first law and we are not showing the Functor mappings between functions, e.g. f —F→ $f_{\uparrow F}$ —G→ $(f_{\uparrow F})_{\uparrow G}$

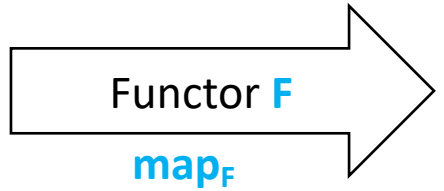
Functions: f, g, g ∘ f, $f_{\uparrow F}$, $g_{\uparrow F}$, $g_{\uparrow F} \circ f_{\uparrow F}$, etc
Higher Order functions: map_F, map_G, $map_{G \circ F}$

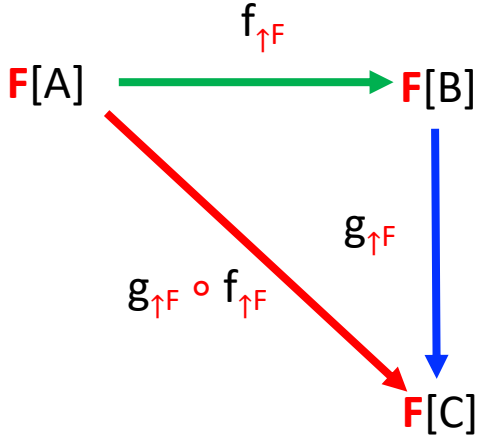












1st Functor Law

```
assert( (mapList(inc compose twice))(List(1,2,3)) == (mapList(inc))(mapList(twice)(List(1,2,3))) )

assert( (mapList(inc compose twice))(List(1,2,3)) == List(3,5,7) )

assert( (mapList(inc))(mapList(twice)(List(1,2,3))) == List(3,5,7) )
```

```
def inc: Int => Int = _ + 1
def twice: Int => Int = _ * 2
```

Mapping the **composition** of **f** and **g** is the same as first applying the **mapping** of **f** and then applying the **mapping** of **g**.

Functor Composition

The **composition** of two **functors** is a **functor** whose **map** is the **composition** of the corresponding **map**s

```
// mapOptionList = mapOption compose mapList

assert(mapOptionList(square)(Some(List(1,2,3))) == mapOption(mapList(square))(Some(List(1,2,3))))

assert(mapOptionList(square)(Some(List(1,2,3))) == Some(List(1,4,9)))

assert((mapOption(mapList(square))(Some(List(1,2,3))) == Some(List(1,4,9)))
```

Mapping f with the **composition** of two **functors** is the same as first **mapping f** with the 1st **functor** and then **mapping** the result with the 2nd **functor**.

1st Functor Law and Functor Composition together

```
// mapOptionList = mapOption compose mapList

assert(((mapOptionList(inc compose twice))(Some(List(1,2,3))))
         == (mapOption(mapList(inc)))((mapOption(mapList(twice)))(Some(List(1,2,3)))))

assert((mapOption(mapList(inc)))((mapOption(mapList(twice)))(Some(List(1,2,3)))) == Some(List(3,5,7)))

assert(((mapOptionList(inc compose twice))(Some(List(1,2,3)))) == Some(List(3,5,7)))
```

Mapping the **composition** of **g** and **f** with the **composition** of two **functors** is the same as doing the following:

1. **mapping f** first with the 1st **functor** and then with the 2nd **functor**
2. **mapping g** first with the 1st **functor** and then with the 2nd **functor**
3. applying first the function computed in (1) and then the function computed in (2)

In **Scalaz** there is a **CompositionFunctor** trait

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```scala
package scalaz

private trait CompositionFunctor[F[_], G[_]] extends Functor[λ[α => F[G[α]]]] {
  implicit def F: Functor[F]
  implicit def G: Functor[G]
  override def map[A, B](fga: F[G[A]])(f: A => B): F[G[B]] =
    F(fga)(G.lift(f))
}
```

Let's adopt a similar approach to provide a **functor composition** trait

```scala
trait CompositionFunctor[F[_],G[_]] extends Functor[λ[α => F[G[α]]]] {
  implicit def F: Functor[F]
  implicit def G: Functor[G]
  override def map[A, B](fga: F[G[A]])(f: A => B): F[G[B]] =
    F.map(fga)(ga => G.map(ga)(f))
}
```

Let's see two examples of **composing functors** using our `CompositionFunctor`

@philip_schwarz

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

```scala
implicit val listFunctor = new Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
}

implicit val optionFunctor = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa map f
}
```

```scala
trait CompositionFunctor[F[_],G[_]] extends Functor[λ[α => F[G[α]]]] {
  implicit def F: Functor[F]
  implicit def G: Functor[G]
  override def map[A, B](fga: F[G[A]])(f: A => B): F[G[B]] =
    F.map(fga)(ga => G.map(ga)(f))
}
```

using https://github.com/non/kind-projector
allows us to simplify type lambda `({type f[α] = F[G[α]]})#f`
to this: `λ[α => F[G[α]]]`

```scala
val listOptionFunctor = new CompositionFunctor[List,Option] {
  implicit def F: Functor[List] = listFunctor
  implicit def G: Functor[Option] = optionFunctor
}
```

Composing the **List Functor** with the **Option Functor**.

```scala
val twice: Int => Int = _ * 2
val listOfOption = List(Some(3),None,Some(4))
val doubledListOfOption = List(Some(6),None,Some(8))

assert(listOptionFunctor.map(listOfOption)(twice) == doubledListOfOption)
```

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

```scala
implicit val listFunctor = new Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
}

import scala.concurrent.ExecutionContext.Implicits.global
val futureFunctor = new Functor[Future] {
  def map[A, B](fa: Future[A])(f: A => B): Future[B] = fa map f (global)
}
```

```scala
trait CompositionFunctor[F[_],G[_]] extends Functor[λ[α => F[G[α]]]] {
  implicit def F: Functor[F]
  implicit def G: Functor[G]
  override def map[A, B](fga: F[G[A]])(f: A => B): F[G[B]] =
    F.map(fga)(ga => G.map(ga)(f))
}
```

```scala
val futureOptionFunctor = new CompositionFunctor[Future,Option] {
  implicit def F: Functor[Future] = futureFunctor
  implicit def G: Functor[Option] = optionFunctor
}
```

Composing the Option Functor with the Future Functor.

```scala
assert( Await.result( futureOptionFunctor.map(Future{ Some(3) })(twice), Duration.Inf) == Some(6) )
```

Instead of having a trait, let's have a **compose** method on **Functor**

@philip_schwarz

```scala
trait Functor[F[_]] {

  def map[A,B](fa: F[A])(f: A => B): F[B]

  def compose[G[_]](G:Functor[G]):Functor[λ[α=>F[G[α]]]] = {
    val self = this
    new Functor[λ[α => F[G[α]]]] {
      override def map[A, B](fga:F[G[A]])(f:A=>B):F[G[B]] =
        self.map(fga)(ga => G.map(ga)(f))
    }
  }
}
```

So that we can create a **composite functor** with

```scala
listFunctor compose optionFunctor
```

rather than with

```scala
new CompositionFunctor[Future[Option]]{
    …
    …
}
```

```scala
implicit val listFunctor = new Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
}

implicit val optionFunctor = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa map f
}
```

```scala
val double: Int => Int = _ * 2
```

```scala
// Functor[List[Option]] = Functor[List] compose Functor[Option]
val optionListFunctor = listFunctor compose optionFunctor

assert(optionListFunctor.map(List(Some(1),Some(2),Some(3)))(double) == List(Some(2),Some(4),Some(6)))
```

```scala
// Functor[Option[List]] = Functor[Option] compose Functor[List]
val listOptionFunctor = optionFunctor compose listFunctor

assert(listOptionFunctor.map(Some(List(1,2,3)))(double) == Some(List(2,4,6)))
```