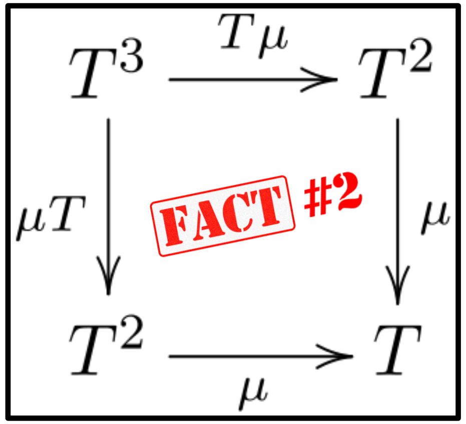
MONAD FACT #2

equivalence of **nested flatMaps** and **chained flatMaps**
for **Kleisli arrow composition**



slides by

@philip_schwarz

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

@philip_schwarz

Kleisli arrows are functions of types like `A => M[B]`, where `M` is a **monadic type constructor**.

Consider three **Kleisli arrows f**, **g** and **h**:

```
f: A => M[B]
g: B => M[C]
h: C => M[D]
```

A convenient way of composing **f**, **g** and **h** in **Scala** is by using a **for comprehension**:

```
for {
  b <- f(a)
  c <- g(b)
  d <- h(c)
} yield d
```

On the next slide we look at a concrete (if contrived) example.

```scala
case class Company(name: String)
case class Driver(name: String)
case class Car(registration: String)
```

Our domain consists of **companies**, **drivers** and **cars**.

```scala
val ibmCompany = Company(name="IBM")
val axaCompany = Company(name="AXA")
val driverJohnSmith = Driver(name="John Smith")
val carRegisteredABC123 = Car(registration="ABC123")
```

Here are a multinational information technology **company**, an **insurance** company, a **driver** and a **car**.

```scala
val driverByCompany = Map(          ibmCompany -> driverJohnSmith      )
val carByDriver     = Map(     driverJohnSmith -> carRegisteredABC123 )
val insuranceByCar  = Map( carRegisteredABC123 -> axaCompany          )
```

The CEO of IBM has a **driver** whose **car** is **insured** by AXA.

```scala
val f: Company => Option[Driver]  = company => driverByCompany.get(company)
val g:  Driver => Option[Car]     =  driver =>     carByDriver.get(driver)
val h:     Car => Option[Company] =     car =>  insuranceByCar.get(car)
```

Here are our three **Kleisli arrows**: **f**, **g** and **h**. They return a **monad** whose **type constructor** is Option.

```scala
def fgh: Company => Option[Company] = company =>
  for {
    driver    <- f(company)
    car       <- g(driver)
    insurance <- h(car)
  } yield insurance
```

Here is function **fgh**, a **Kleisli arrow** that is the **composition** of **f**, **g** and **h**.

Better names for **Kleisli arrows f**, **g** and **h** could be the following:

- getCEODriverOf(company)
- getCarDrivenBy(driver)
- getInsurerFor(car)

But for our purposes we can forget what the functions are computing and just concentrate on the fact that they are **Kleisli arrows**.

```scala
assert( fgh(ibmCompany) == Some(Company("AXA")) )
assert( fgh(axaCompany) == None )
```

Here we test that the insurer of the car driven by the driver of IBM's CEO is AXA.
There is no insurer of the car driven by the driver of AXA's CEO (no such car nor driver).

As we saw in **MONAD FACT #1**, the code on the left **desugars** to the code on the right

```
val fgh : Company => Option[Company] = company =>
  for {
    driver    <- f(company)
    car       <- g(driver)
    insurance <- h(car)
  } yield insurance
```

desugars to

```
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver)
      .flatMap { car => h(car)
        .map { insurance => insurance
        }
      }
    }
```

Mapped function **insurance => insurance** is just the **identity function**

The reason why a **monad** has a **map** function is that every **monad** is also a **functor**. The **map** function of a **functor** is subject to the following **functor law**:

```
map(x)(identity) == x.
```

Thanks to this law, we can **simplify our code** by dropping the invocation of **map**.

```
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver)
      .flatMap { car => h(car)
        .map { identity
        }
      }
    }
```

simplified

```
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver)
      .flatMap { car => h(car)
      }
    }
```

At this point we can go one of two ways.

```scala
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver)
      .flatMap { car => h(car)
      }
    }
```

```scala
assert( fgh(ibmCompany) == Some(Company("AXA")) )
assert( fgh(axaCompany) == None )
```

We can **simplify the above** just a little by making it **less verbose**

```scala
val fgh : Company => Option[Company] = company =>
  f(company) flatMap { driver => g(driver) flatMap h }
```

Later on I'll refer to this as **the nested flatMap function**.

@philip_schwarz

Or alternatively, it turns out that (see later) we can **rearrange the flatMap invocations so that rather than being nested, they are chained**.

```scala
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver)
      .flatMap { car => h(car)
      }
    }
```

from **nested flatMaps** to **chained flatMaps**

```scala
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver) }
    .flatMap { car => h(car) }
```

And finally, we can **simplify** this a bit

by making it **less verbose**

```
val fgh : Company => Option[Company] = company =>
  f(company)
    .flatMap { driver => g(driver) }
    .flatMap { car => h(car) }
```

simplified

```
val fgh: Company => Option[Company] = company =>
  f(company) flatMap g flatMap h
```

On the next slide, I'll be referring to the above function as **the chained flatMap function**

We started off with the **for comprehension function** on the right

```
val fgh : Company => Option[Company] = company =>
  for {
    driver    <- f(company)
    car       <- g(driver)
    insurance <- h(car)
  } yield insurance
```

for comprehension function

And we refactored it into the **two equivalent desugared functions** below

chained flatMap function

```
val fgh: Company => Option[Company] = company =>
  f(company) flatMap g flatMap h
```

nested flatMap function

```
val fgh : Company => Option[Company] = company =>
  f(company) flatMap { driver => g(driver) flatMap h }
```

@philip_schwarz

The two **desugared** versions of the **for comprehension function** are **equivalent** because the **flatMap** function of a **monad** is subject to a **monadic Law of Associativity**:

an operation ✻ is **associative** if it doesn't matter whether we parenthesize it ((x ✻ y) ✻ z) or (x ✻ (y ✻ z))

Monadic Law of Associativity

```
(m flatMap f) flatMap g  ≡  m flatMap ( x => f(x) flatMap g )
```

This holds for all values **m, f** and **g** of the appropriate types (see right).

```
m: M[A]
f: A => M[B]
g: B => M[C]
```

While on the left hand side of the equation the invocations of **flatMap** are being **chained**, on the right hand side of the equation the invocations are being **nested**.

e.g. in our example we have the following:

```
(f(a) flatMap g) flatMap h  ≡  f(a) flatMap ( b => g(b) flatMap h )
```

```
f: A => M[B]
g: B => M[C]
h: C => M[D]
```

Here is a recap

```
assert(
    // desugars to nested flatMaps and map
    ((company: Company) =>
      for {
        driver    <- f(company)
        car       <- g(driver)
        insurance <- h(car)
      } yield insurance)
    (ibmCompany)
  ==
    (Some(Company("AXA")))
)
```

Monadic Law of Associativity

```
(m flatMap f) flatMap g  ≡  m flatMap ( x => f(x) flatMap g )
```

```
m: M[A]
f: A => M[B]
g: B => M[C]
```

The **Monadic Law of Associativity** in the context of this example

```
(f(a) flatMap g) flatMap h  ≡  f(a) flatMap ( b => g(b) flatMap h )
```

```
f: A => M[B]
g: B => M[C]
h: C => M[D]
```

```
assert(
    ((company: Company) =>
      for {
        driver    <- f(company)
        car       <- g(driver)
        insurance <- h(car)
      } yield insurance)
    (ibmCompany)
  ==
    // chained flatMaps
    ((company: Company) => f(company) flatMap g flatMap h )
    (ibmCompany)
)
```

```
assert(
    ((company: Company) =>
      for {
        driver    <- f(company)
        car       <- g(driver)
        insurance <- h(car)
      } yield insurance)
    (ibmCompany)
  ==
    // nested flatMaps
    ((company:Company) => f(company) flatMap { driver => g(driver) flatMap h } )
    (ibmCompany)
)
```

```
assert(
    ((company:Company) => f(company) flatMap g flatMap h)(ibmCompany) // chained flatMaps
  == ((company:Company) => f(company) flatMap { driver => g(driver) flatMap h } )(ibmCompany) // nested flatMaps
)
```

See the following for the list of all available slide decks in the **MONAD FACT** series

slideshare https://www.slideshare.net/pjschwarz/the-monad-fact-slide-deck-series