Zrozumieć monady

Grzegorz Balcerek

2015-04-29

```
val sqrt = math.sqrt _
val acos = math.acos _
val divTen = (x:Double) => x/10.0
```

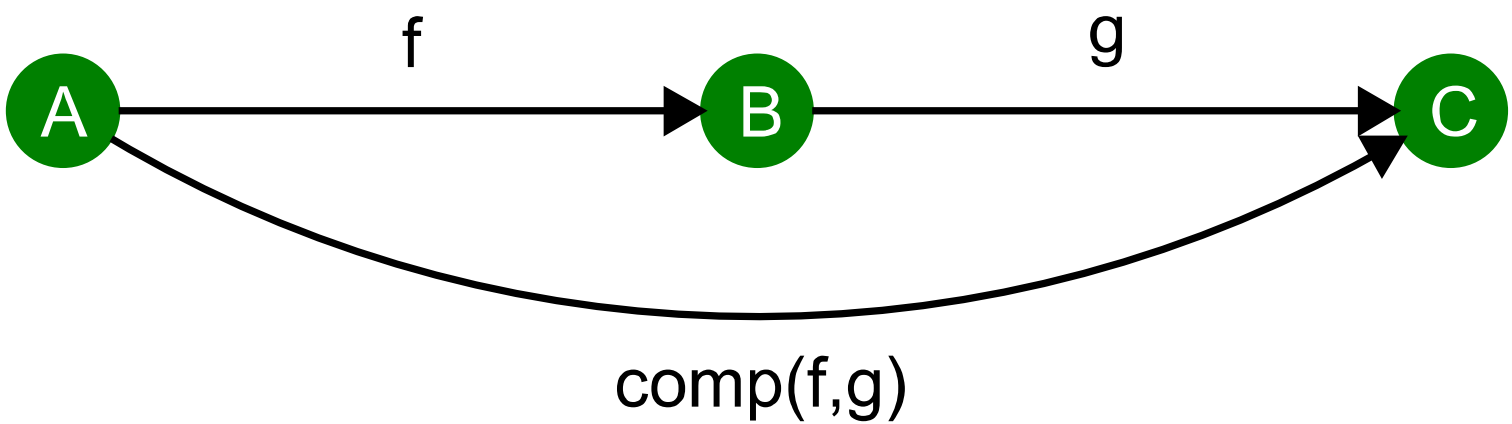
```
val sqrt = math.sqrt _
val acos = math.acos _
val divTen = (x:Double) => x/10.0

scala> sqrt(64.0)
res0: Double = 8.0
```

```
val sqrt = math.sqrt _
val acos = math.acos _
val divTen = (x:Double) => x/10.0

scala> sqrt(64.0)
res0: Double = 8.0

scala> divTen(8.0)
res1: Double = 0.8
```

```
val sqrt = math.sqrt _
val acos = math.acos _
val divTen = (x:Double) => x/10.0

scala> sqrt(64.0)
res0: Double = 8.0

scala> divTen(8.0)
res1: Double = 0.8

scala> acos(0.8)
res2: Double = 0.6435011087932843
```

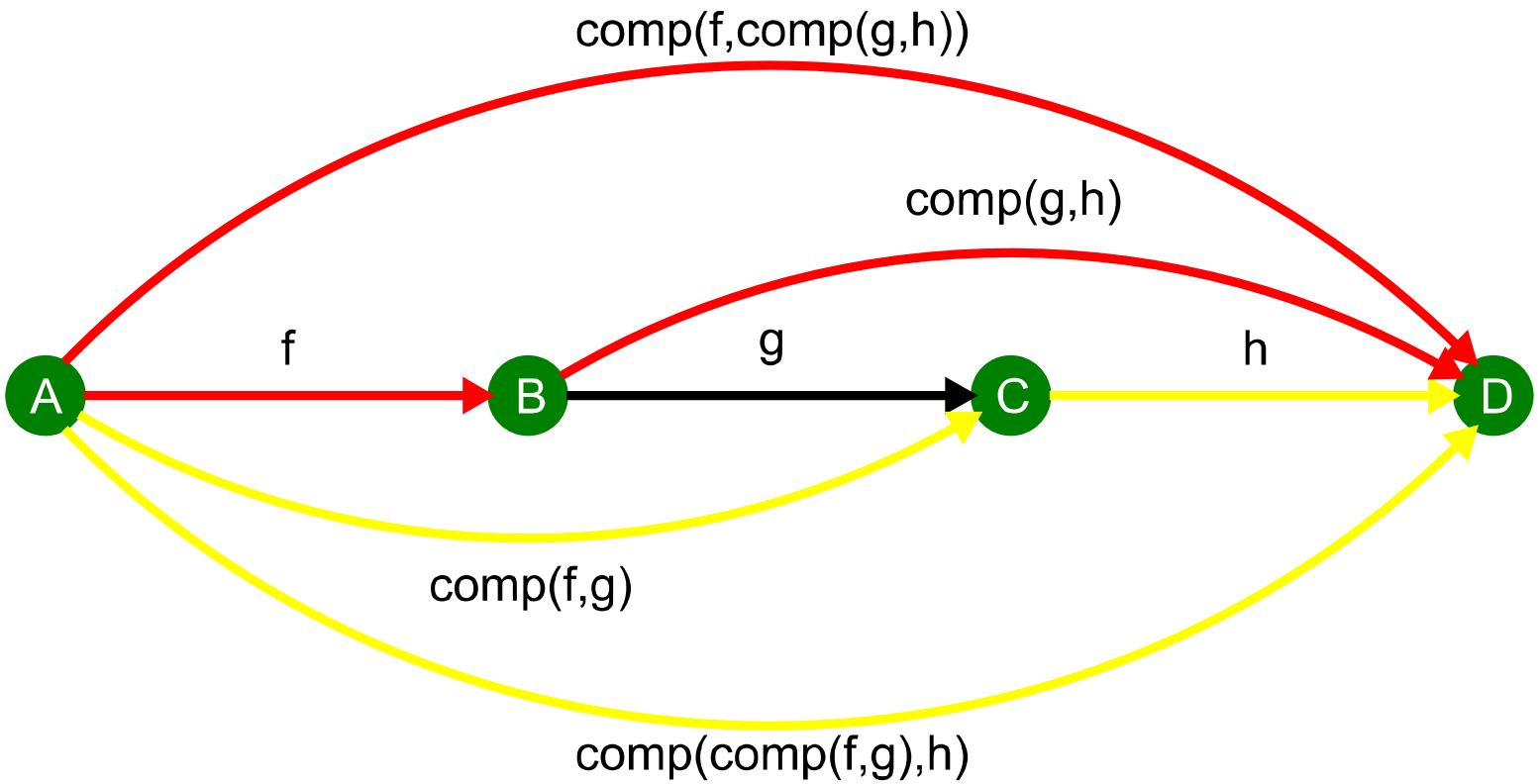
```
def comp[A,B,C](
  f: A => B, g: B => C): A => C =
    (a) => g(f(a))
```

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



```
comp(f,comp(g,h)) == comp(comp(f,g),h)
```

```
comp(f,comp(g,h)) == comp(comp(f,g),h)
```

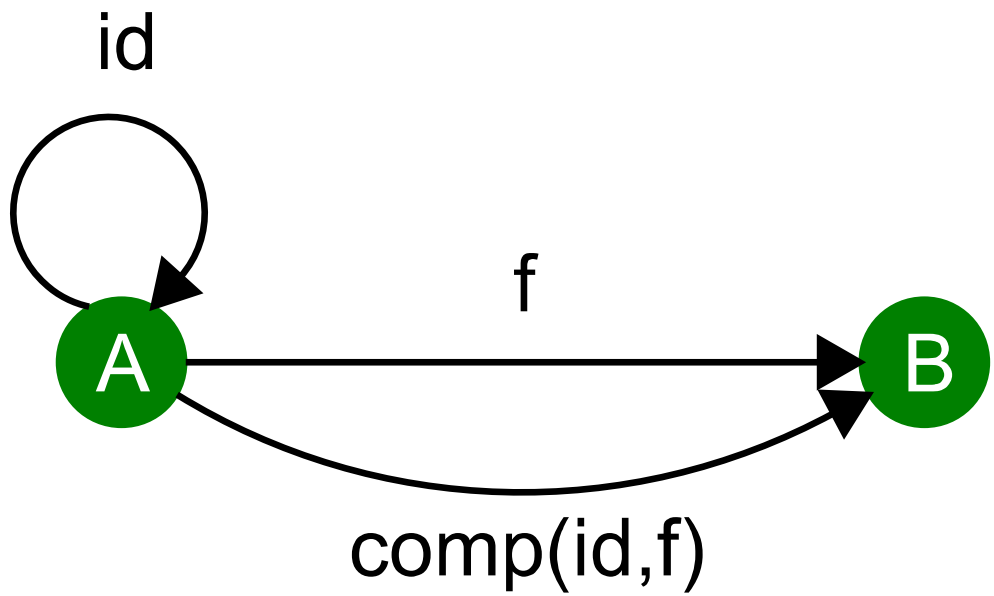


```
def id[A](a:A) = a
```

```
def id[A](a:A) = a

scala> id(1)
res3: Int = 1
```

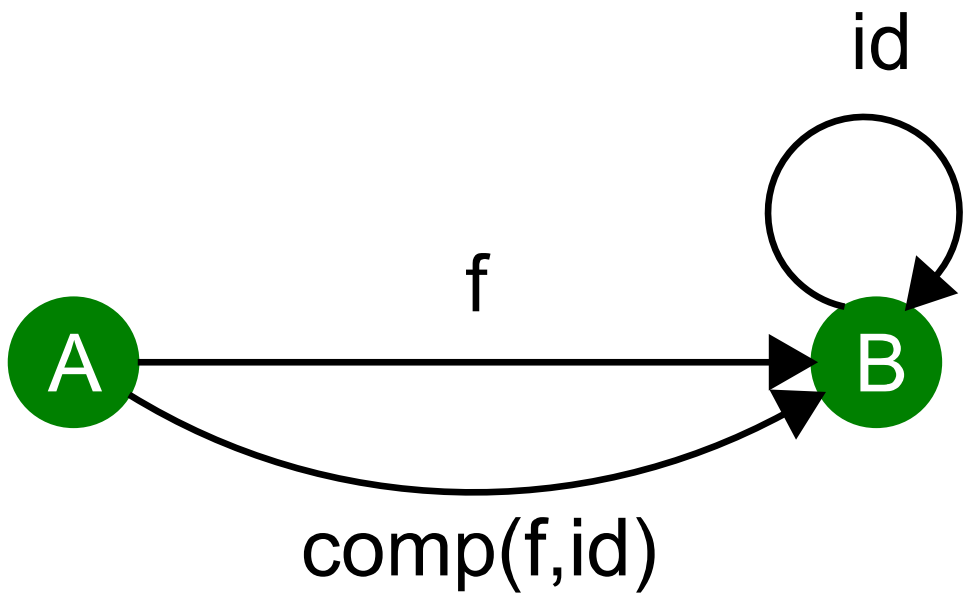
```
comp(id,f) == f
```

```
comp(id,f) == f
```



```
comp(f,id) == f
```

```
comp(f,id) == f
```



```
def compSeq[A](fs: (A => A)*): A => A =
  fs.foldRight[A=>A](id)(comp)
```

```
def compSeq[A](fs: (A => A)*): A => A =
  fs.foldRight[A=>A](id)(comp)

val f = compSeq(sqrt,divTen,acos)
```

```
def compSeq[A](fs: (A => A)*): A => A =
  fs.foldRight[A=>A](id)(comp)

val f = compSeq(sqrt,divTen,acos)

scala> f(64.0)
res4: Double = 0.6435011087932843
```

```
scala> sqrt(-1.0)
res5: Double = NaN
```

```
scala> sqrt(-1.0)
res5: Double = NaN

scala> acos(10.0)
res6: Double = NaN
```

```
scala> sqrt(-1.0)
res5: Double = NaN

scala> acos(10.0)
res6: Double = NaN

scala> f(200.0)
res7: Double = NaN
```

```
val sqrt0 = (x:Double) =>
  if (x >= 0.0) Some(sqrt(x)) else None
```

```
val sqrtO = (x:Double) =>
  if (x >= 0.0) Some(sqrt(x)) else None

val divTenO = (x:Double) =>
  Option(divTen(x))
```

```
val sqrtO = (x:Double) =>
  if (x >= 0.0) Some(sqrt(x)) else None

val divTenO = (x:Double) =>
  Option(divTen(x))

val acosO = (x:Double) =>
  if (x >= -1.0 && x <= 1.0)
    Some(acos(x)) else None
```

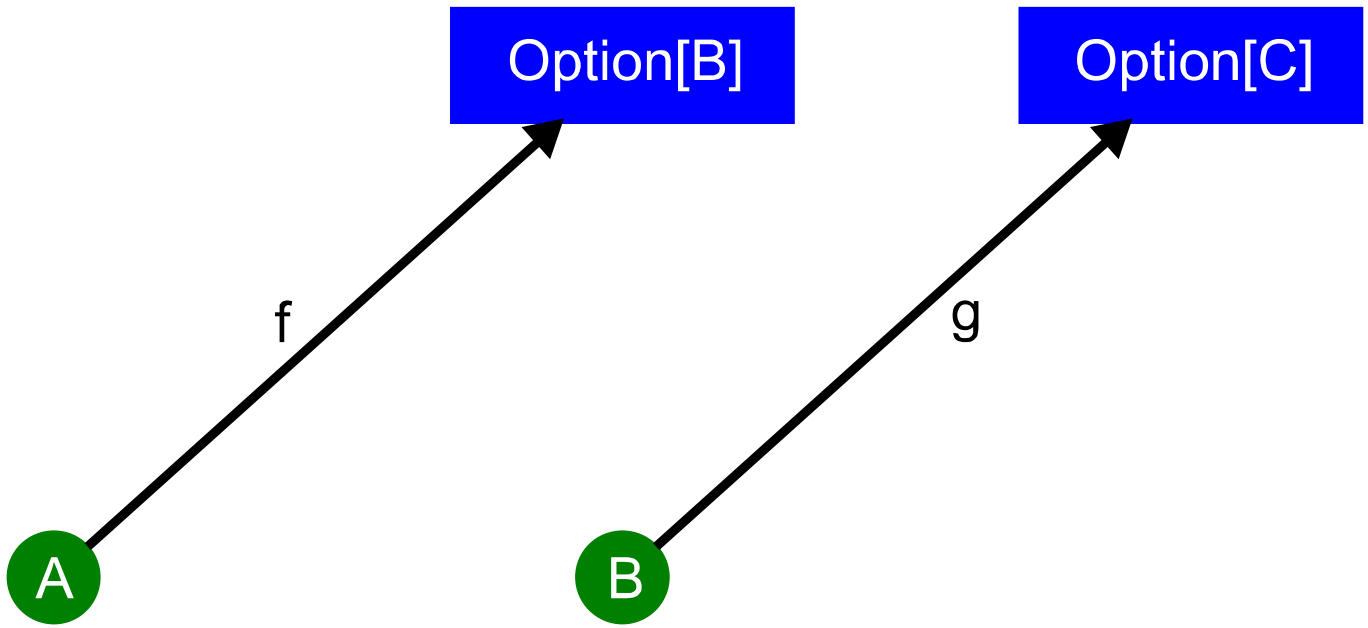
```
scala> sqrtO(64.0)
res8: Option[Double] = Some(8.0)
```

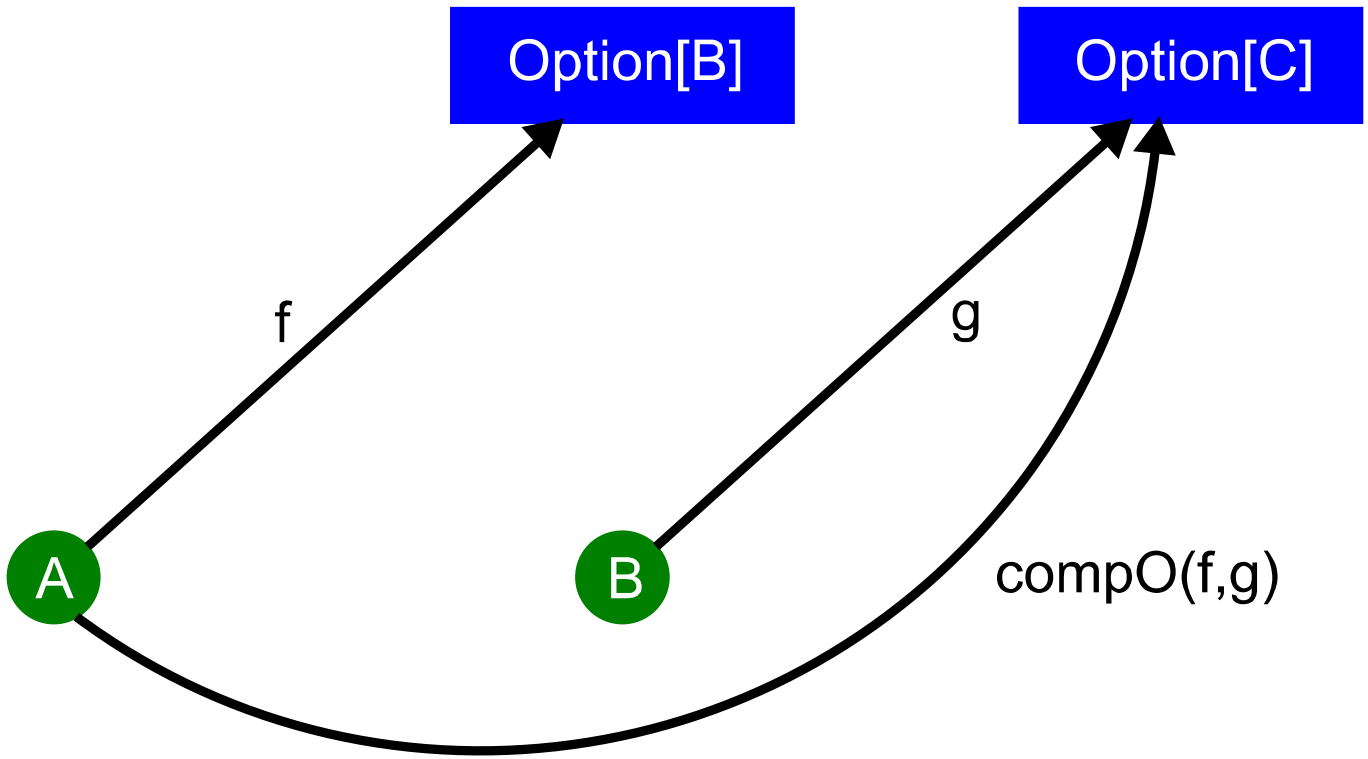
```
scala> sqrtO(64.0)
res8: Option[Double] = Some(8.0)

scala> divTenO(8.0)
res9: Option[Double] = Some(0.8)
```

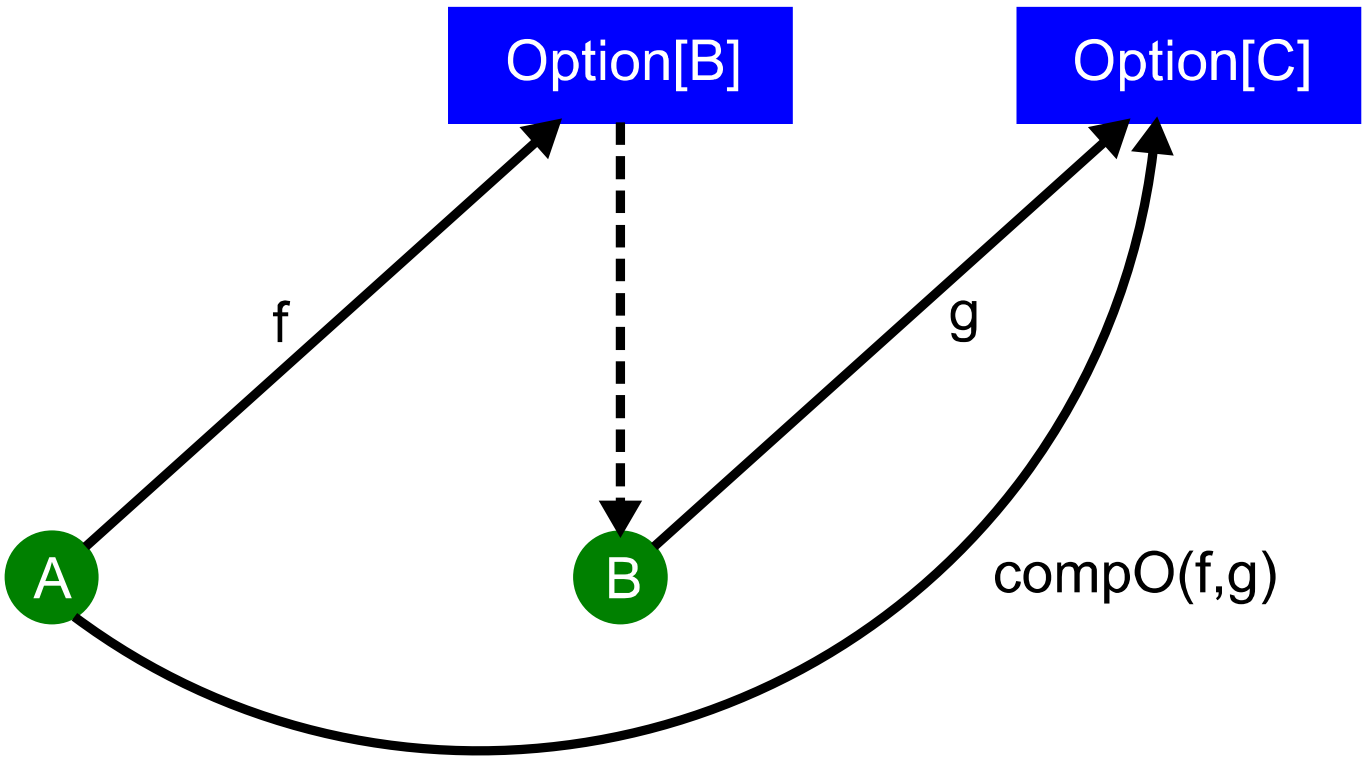
```
scala> sqrtO(64.0)
res8: Option[Double] = Some(8.0)

scala> divTenO(8.0)
res9: Option[Double] = Some(0.8)

scala> acosO(0.8)
res10: Option[Double] = Some(0.6435011087932843)
```

```
scala> sqrtO(64.0)
res8: Option[Double] = Some(8.0)

scala> divTenO(8.0)
res9: Option[Double] = Some(0.8)

scala> acosO(0.8)
res10: Option[Double] = Some(0.6435011087932843)

scala> sqrtO(-1.0)
res11: Option[Double] = None
```

```
scala> sqrtO(64.0)
res8: Option[Double] = Some(8.0)

scala> divTenO(8.0)
res9: Option[Double] = Some(0.8)

scala> acosO(0.8)
res10: Option[Double] = Some(0.6435011087932843)

scala> sqrtO(-1.0)
res11: Option[Double] = None

scala> acosO(10.0)
res12: Option[Double] = None
```


Option[B]
Option[C]
f
g
A
B


Option[B]
Option[C]
f
g
A
B
compO(f,g)


Option[B]
Option[C]
f
g
A
B
compO(f,g)

```scala
def compO[A,B,C](
      f: A => Option[B],
      g: B => Option[C]): A => Option[C] =
  (a) => f(a) match {
           case None => None
           case Some(b) => g(b)
         }
```

```scala
def idO[A](a: A): Option[A] = Some(a)
```

```
def idO[A](a: A): Option[A] = Some(a)

scala> idO(1)
res13: Option[Int] = Some(1)
```

```scala
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)
```

```
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)

val fO = compOSeq(sqrtO,divTenO,acosO)
```

```
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)

val fO = compOSeq(sqrtO,divTenO,acosO)

scala> fO(64.0)
res14: Option[Double] = Some(0.6435011087932843)
```

```
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)

val fO = compOSeq(sqrtO,divTenO,acosO)

scala> fO(64.0)
res14: Option[Double] = Some(0.6435011087932843)

scala> fO(200.0)
res15: Option[Double] = None
```

```
val sqrtL = (x:Double) =>
  if (x >= 0.0)
    List(sqrt(x),-sqrt(x)) else Nil
```

```
val sqrtL = (x:Double) =>
  if (x >= 0.0)
    List(sqrt(x),-sqrt(x)) else Nil

val divTenL = (x:Double) =>
  List(divTen(x))
```

```
val sqrtL = (x:Double) =>
  if (x >= 0.0)
    List(sqrt(x),-sqrt(x)) else Nil

val divTenL = (x:Double) =>
  List(divTen(x))

val acosL = (x:Double) =>
  if (x >= -1.0 && x <= 1.0)
    List(acos(x)) else Nil
```

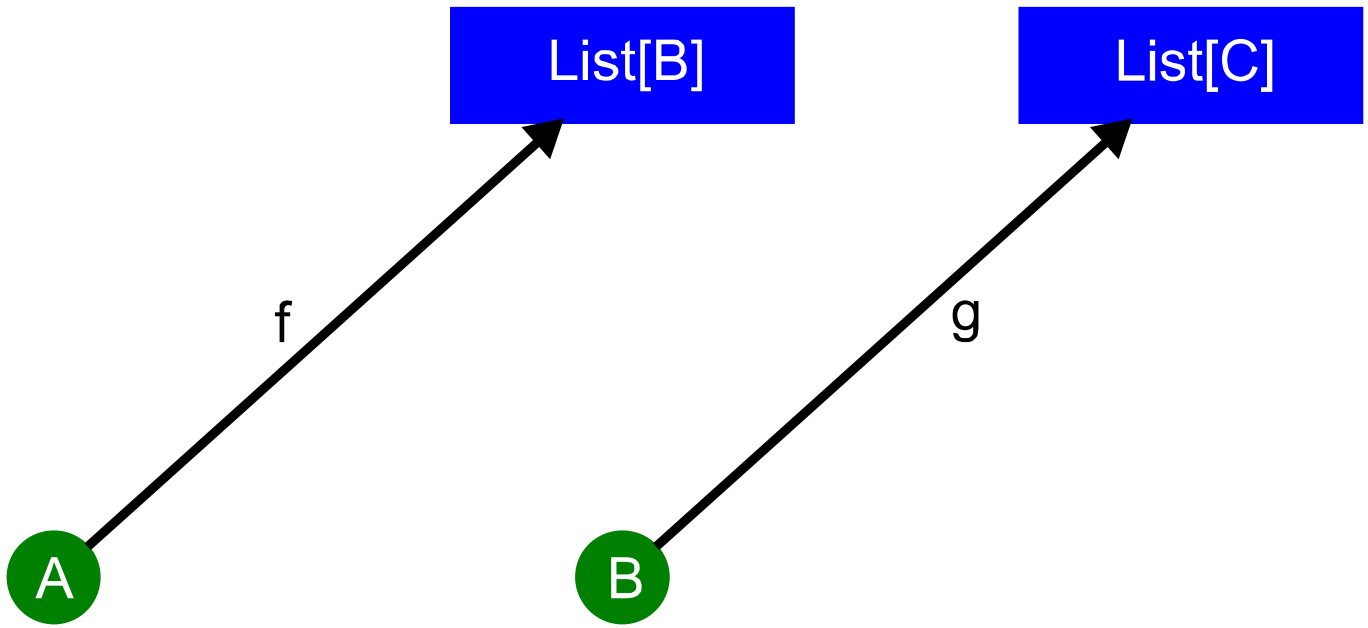
```
scala> sqrtL(64.0)
res16: List[Double] = List(8.0, -8.0)
```

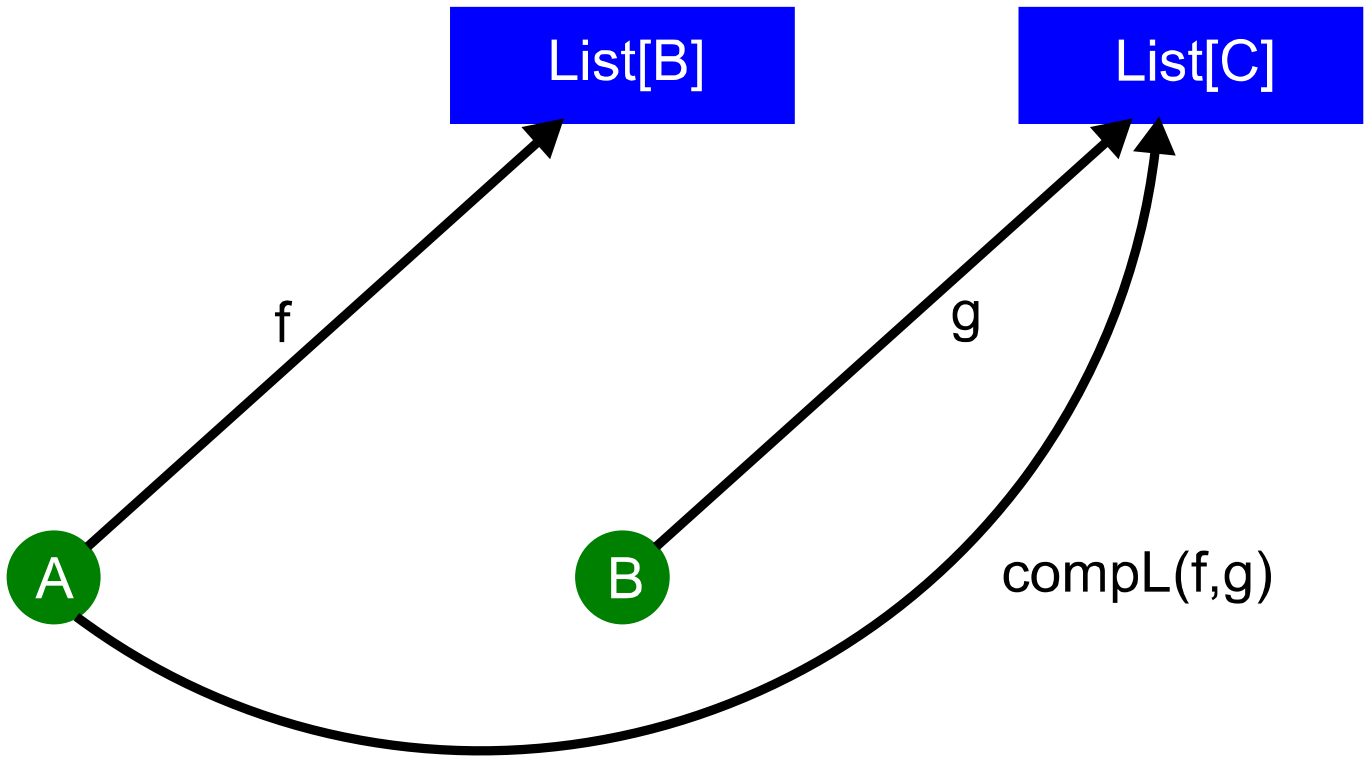
```
scala> sqrtL(64.0)
res16: List[Double] = List(8.0, -8.0)

scala> acosL(0.8)
res17: List[Double] = List(0.6435011087932
843)
```

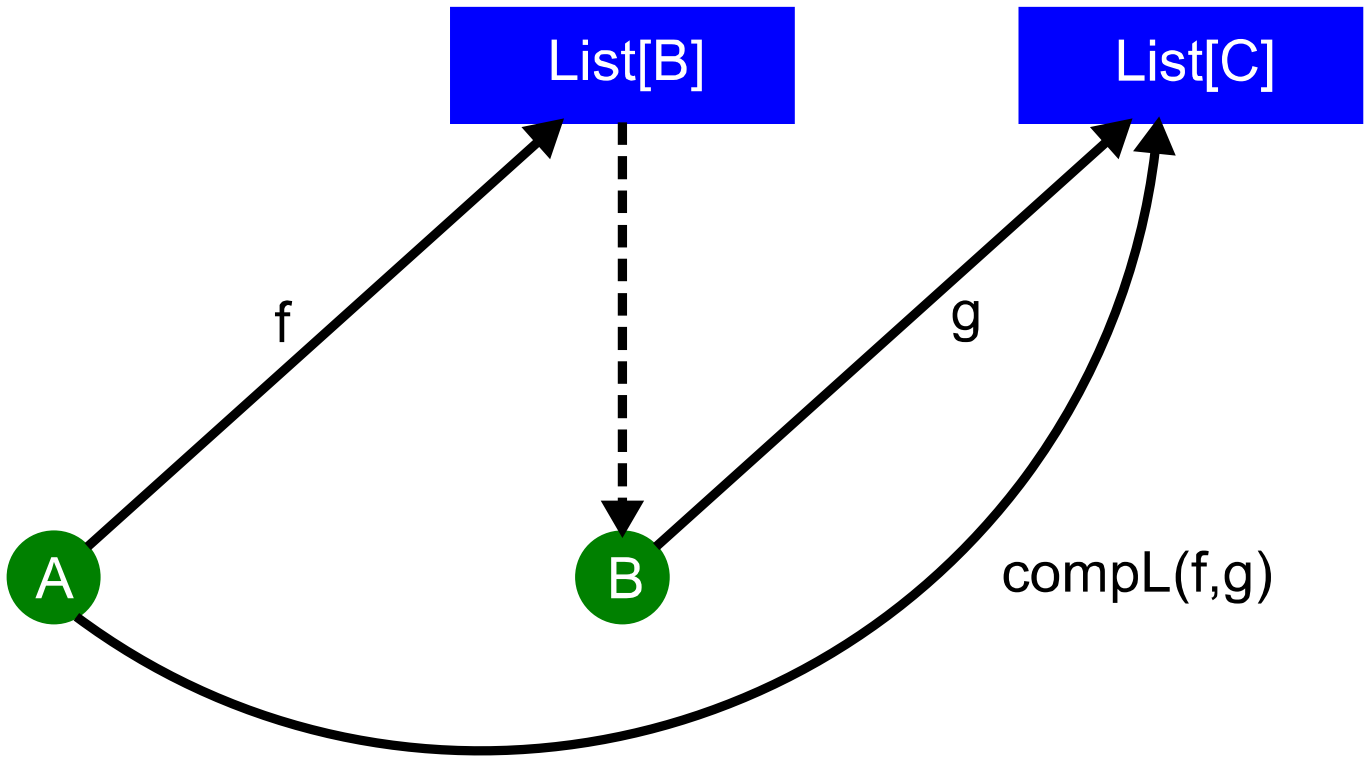
```
scala> sqrtL(64.0)
res16: List[Double] = List(8.0, -8.0)

scala> acosL(0.8)
res17: List[Double] = List(0.6435011087932
843)

scala> acosL(-0.8)
res18: List[Double] = List(2.4980915447965
09)
```

```
scala> sqrtL(64.0)
res16: List[Double] = List(8.0, -8.0)

scala> acosL(0.8)
res17: List[Double] = List(0.6435011087932
843)

scala> acosL(-0.8)
res18: List[Double] = List(2.4980915447965
09)

scala> sqrtL(-1.0)
res19: List[Double] = List()
```

```
scala> sqrtL(64.0)
res16: List[Double] = List(8.0, -8.0)

scala> acosL(0.8)
res17: List[Double] = List(0.6435011087932
843)

scala> acosL(-0.8)
res18: List[Double] = List(2.4980915447965
09)

scala> sqrtL(-1.0)
res19: List[Double] = List()

scala> acosL(10.0)
res20: List[Double] = List()
```


List[B]
List[C]
f
g
A
B


List[B]
List[C]
f
g
A
B
compL(f,g)


List[B]
List[C]
f
g
A
B
compL(f,g)

```
def compL[A,B,C](
      f: A => List[B],
      g: B => List[C]): A => List[C] =
  (a) => {
    val bs = f(a)
    var c = List[C]()
    bs.foreach(b => c = c ++ g(b))
    c
  }
```

```scala
def idL[A]: A => List[A] = List(_)
```

```
def idL[A]: A => List[A] = List(_)

scala> idL(1)
res21: List[Int] = List(1)
```

```
def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)
```

```
def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)

val fL = compLSeq(sqrtL,divTenL,acosL)
```

```
def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)

val fL = compLSeq(sqrtL,divTenL,acosL)

scala> fL(64.0)
res22: List[Double] = List(0.6435011087932
843, 2.498091544796509)
```

```
def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)

val fL = compLSeq(sqrtL,divTenL,acosL)

scala> fL(64.0)
res22: List[Double] = List(0.6435011087932
843, 2.498091544796509)

scala> fL(200.0)
res23: List[Double] = List()
```

```scala
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)
```

```
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)

def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)
```

```
def compOSeq[A](
  fs: (A => Option[A])*): A => Option[A] =
    fs.foldRight[A => Option[A]](idO)(compO)

def compLSeq[A](
  fs: (A => List[A])*): A => List[A] =
    fs.foldRight[A => List[A]](idL)(compL)

def composeSeq[A](
  fs: (A => F[A])*): A => F[A] =
    fs.foldRight[A => F[A]](id)(compose)
```

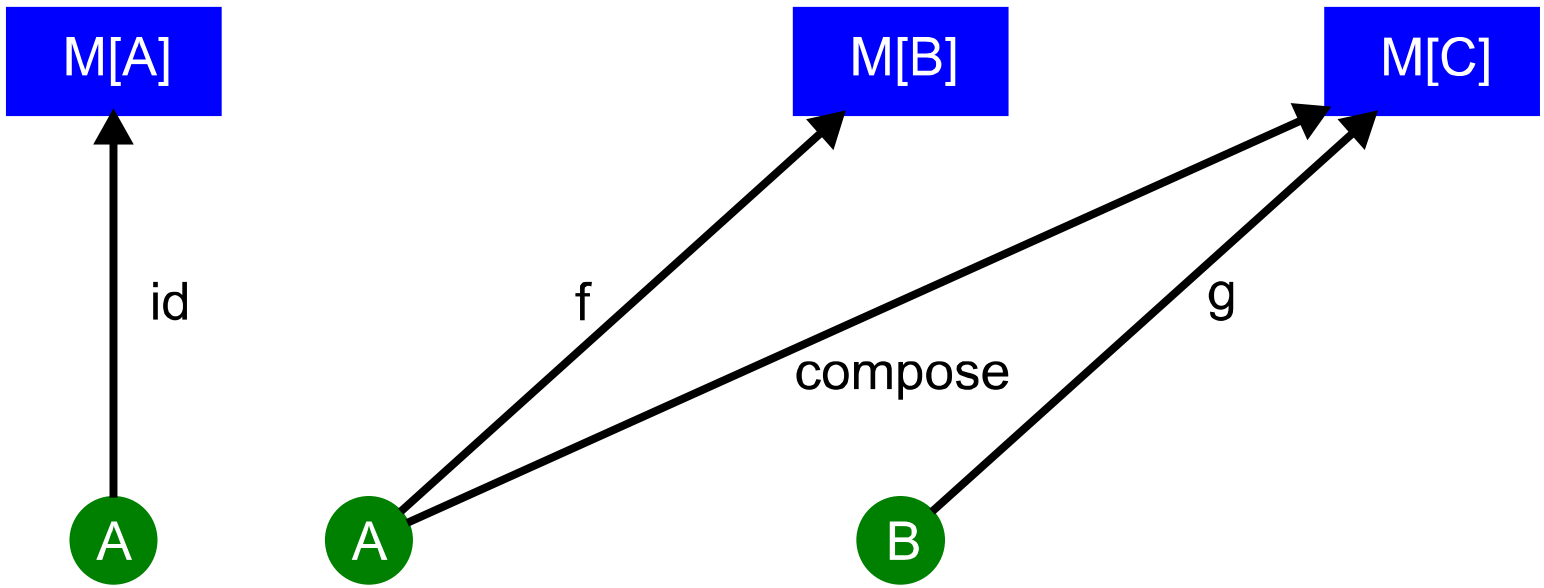
```
trait Compose[F[_]] {
```

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

  def id[A](a: A): F[A]
```

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

  def id[A](a: A): F[A]

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

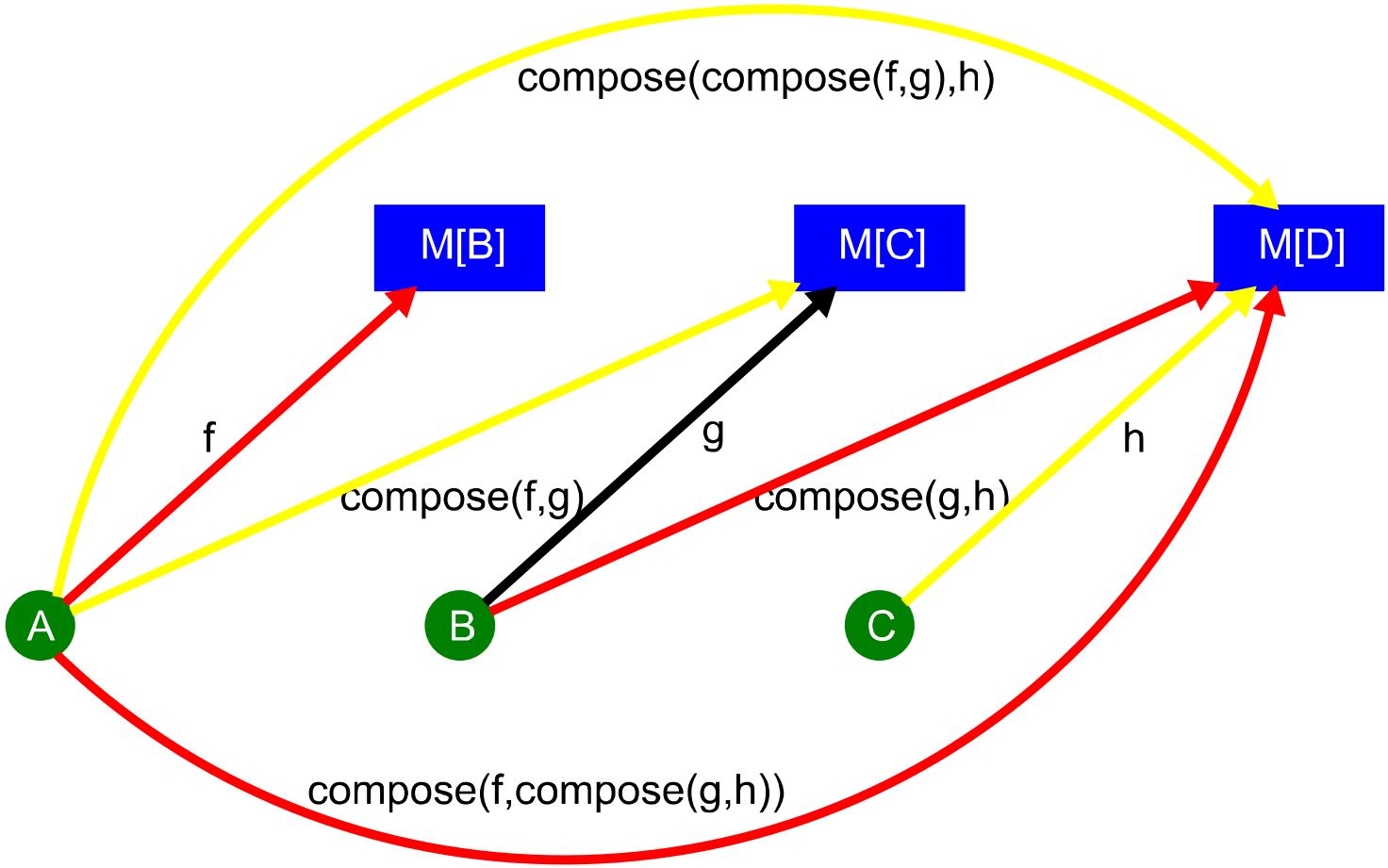
```scala
trait Compose[F[_]] {

  def id[A](a: A): F[A]

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

  def composeSeq[A](
    fs: (A => F[A])*): A => F[A] =
      fs.foldRight[A => F[A]](id)(compose)

}
```

```scala
trait Monad[M[_]] {

  def id[A](a: A): M[A]

  def compose[A,B,C](
    f: A => M[B], g: B => M[C]): A => M[C]

  def composeSeq[A](
    fs: (A => M[A])*): A => M[A] =
      fs.foldRight[A => M[A]](id)(compose)

}
```


M[A]
M[B]
M[C]
id
f
compose
g
A
A
B

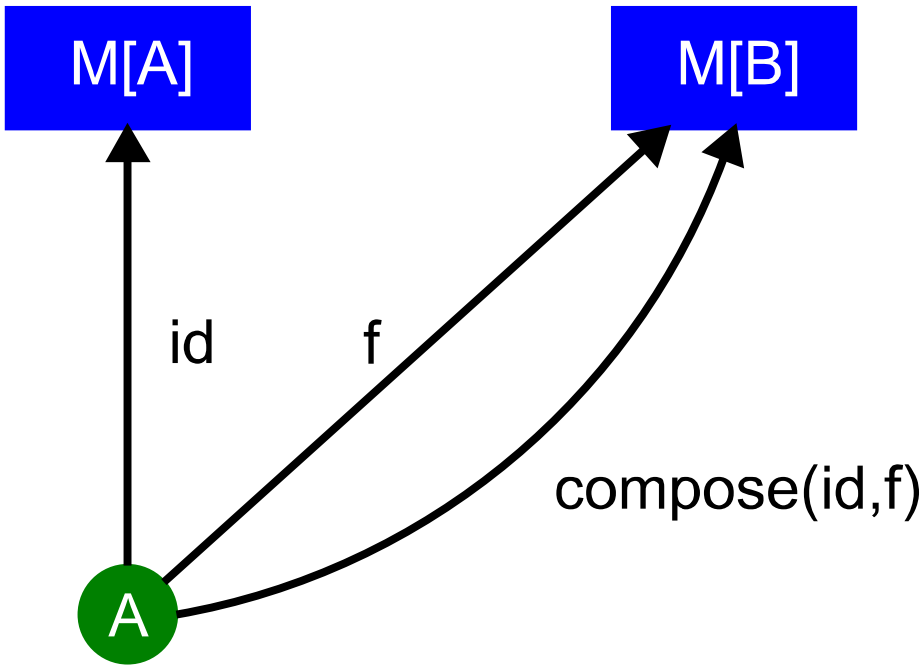
```
compose(compose(f,g),h)
          ==
compose(f,compose(g,h))
```

```
compose(compose(f,g),h)
          ==
compose(f,compose(g,h))
```



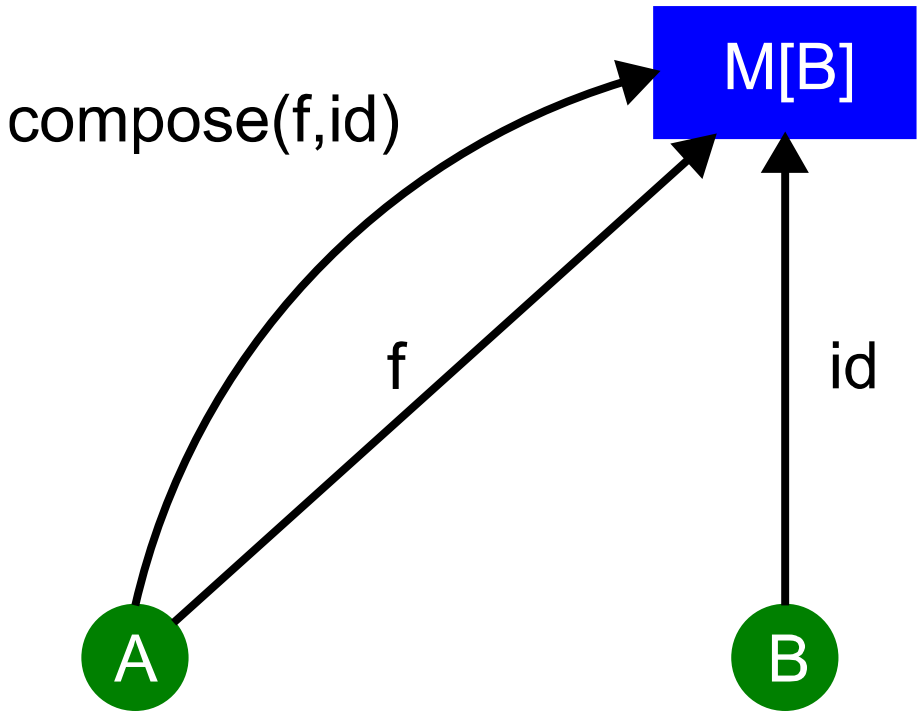
```
compose(id,f) == f
```

```
compose(id,f) == f
```



```
compose(f,id) == f
```

```
compose(f,id) == f
```



Monad:

```
def id[A](a: A): M[A]

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

```
compose(compose(f,g),h)
          ==
compose(f,compose(g,h))

compose(id,f) == f

compose(f,id) == f
```

```
implicit object optionMonad extends
                              Monad[Option] {

  def id[A](a: A): Option[A] = Some(a)

  def compose[A,B,C](
        f: A => Option[B],
        g: B => Option[C]): A => Option[C] =
    (a) => f(a) match {
             case None => None
             case Some(b) => g(b)
           }

}
```

```
def composeSeq[A,M[_]:Monad](
  fs: (A => M[A])*): A => M[A] =
    implicitly[Monad[M]].composeSeq(fs: _*)
```

```
def composeSeq[A,M[_]:Monad](
  fs: (A => M[A])*): A => M[A] =
    implicitly[Monad[M]].composeSeq(fs: _*)

val f20 = composeSeq(sqrt0,divTen0,acos0)
```

```
def composeSeq[A,M[_]:Monad](
  fs: (A => M[A])*): A => M[A] =
    implicitly[Monad[M]].composeSeq(fs: _*)

val f20 = composeSeq(sqrt0,divTen0,acos0)

scala> f20(64.0)
res24: Option[Double] = Some(0.6435011087932843)
```

```
def composeSeq[A,M[_]:Monad](
  fs: (A => M[A])*): A => M[A] =
    implicitly[Monad[M]].composeSeq(fs: _*)

val f20 = composeSeq(sqrt0,divTen0,acos0)

scala> f20(64.0)
res24: Option[Double] = Some(0.6435011087932843)

scala> f20(200.0)
res25: Option[Double] = None
```

```
implicit object listMonad extends
                              Monad[List] {

  def id[A](a: A) = List(a)

  def compose[A,B,C](f: A => List[B],
        g: B => List[C]): A => List[C] =
    (a) => {
      val bs = f(a)
      var c = List[C]()
      bs.foreach(b => c = c ++ g(b))
      c
    }
}
```

```
val f2L = composeSeq(sqrtL,divTenL,acosL)
```

```
val f2L = composeSeq(sqrtL,divTenL,acosL)

scala> f2L(64.0)
res26: List[Double] = List(0.6435011087932
843, 2.498091544796509)
```

```
val f2L = composeSeq(sqrtL,divTenL,acosL)

scala> f2L(64.0)
res26: List[Double] = List(0.6435011087932
843, 2.498091544796509)

scala> f2L(200.0)
res27: List[Double] = List()
```

Monad (1):

Monad (1):

```
def id[A](a: A): M[A]
```

Monad (1):

```
def id[A](a: A): M[A]

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

Monad (1):

```
def id[A](a: A): M[A]

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

----

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]
```

Monad (1):

```
def id[A](a: A): M[A]

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

----

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

def join[A](mma: M[M[A]]): M[A]
```

Monad (1):

```
def id[A](a: A): M[A]

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

----

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

def join[A](mma: M[M[A]]): M[A]

def ap[A,B](ma: M[A], mab: M[A => B]): M[B]
```

Monad (1):

```
def id[A](a: A): M[A]

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

----

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

def join[A](mma: M[M[A]]): M[A]

def ap[A,B](ma: M[A], mab: M[A => B]): M[B]

def map[A,B](ma: M[A], f: A => B): M[B]
```

Monad (2):

Monad (2):

```
def id[A](a: A): M[A]
```

Monad (2):

```
def id[A](a: A): M[A]

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]
```

Monad (2):

```
def id[A](a: A): M[A]

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

----

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

def join[A](mma: M[M[A]]): M[A]

def ap[A,B](ma: M[A], mab: M[A => B]): M[B]

def map[A,B](ma: M[A], f: A => B): M[B]
```

Monad (3):

Monad (3):

```
def id[A](a: A): M[A]
```

Monad (3):

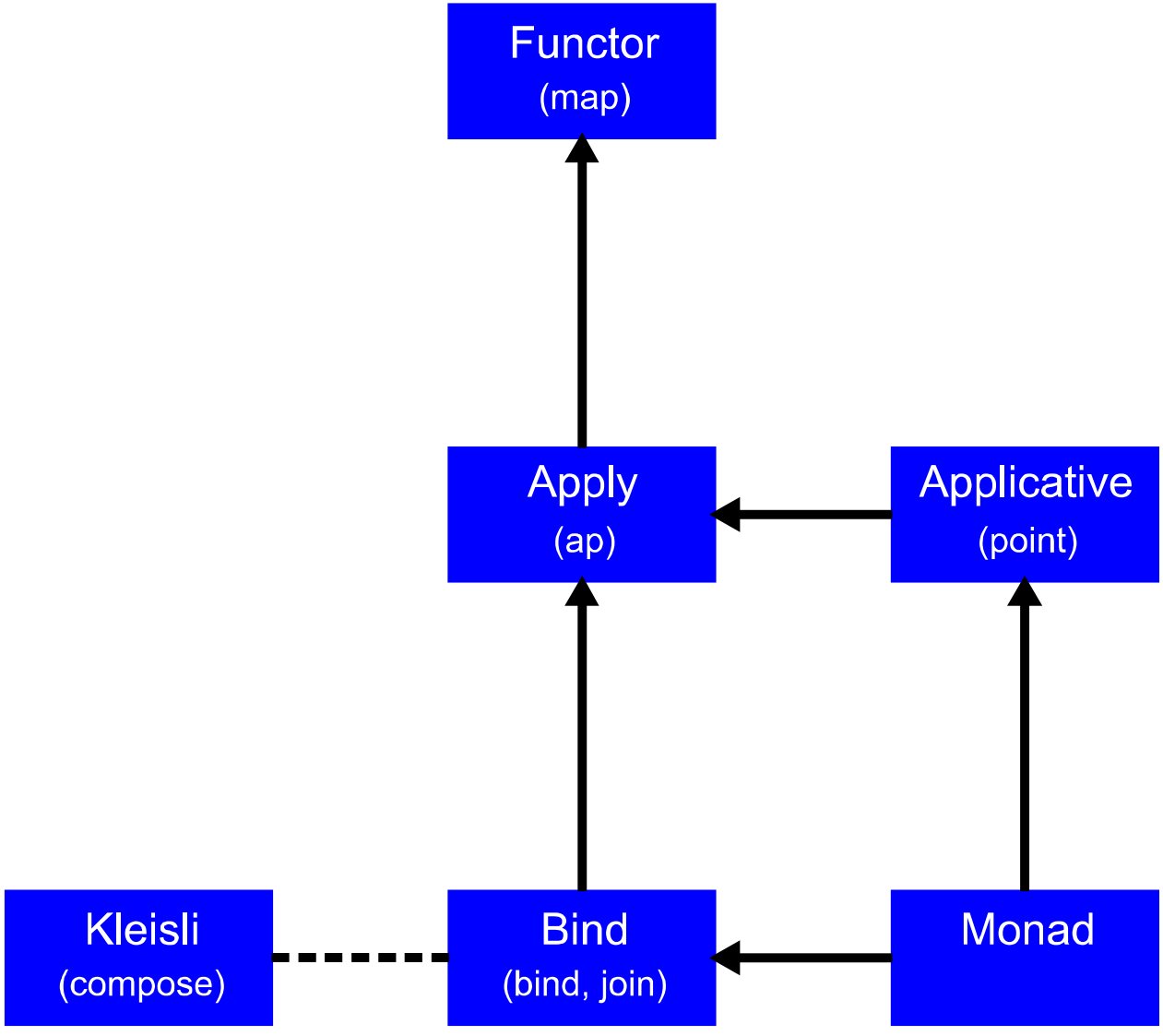
```
def id[A](a: A): M[A]

def map[A,B](ma: M[A], f: A => B): M[B]
```

Monad (3):

```
def id[A](a: A): M[A]

def map[A,B](ma: M[A], f: A => B): M[B]

def join[A](mma: M[M[A]]): M[A]
```

Monad (3):

```
def id[A](a: A): M[A]

def map[A,B](ma: M[A], f: A => B): M[B]

def join[A](mma: M[M[A]]): M[A]

----

def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

def compose[A,B,C](f: A => M[B],
                   g: B => M[C]): A => M[C]

def ap[A,B](ma: M[A], mab: M[A => B]): M[B]
```


Functor
(map)
Apply
(ap)
Applicative
(point)
Kleisli
(compose)
Bind
(bind, join)
Monad

```
def f3O(a:Double) =
  sqrtO(a).flatMap( b =>
  divTenO(b).flatMap( c =>
  acosO(c)))
```

```
def f3O(a:Double) =
  sqrtO(a).flatMap( b =>
  divTenO(b).flatMap( c =>
  acosO(c)))

def f3O(a:Double) =
  sqrtO(a).flatMap( b =>
  divTenO(b).flatMap( c =>
  acosO(c).map( d => d )))
```

```
def f30(a:Double) =
  sqrt0(a).flatMap( b =>
  divTen0(b).flatMap( c =>
  acos0(c)))

def f30(a:Double) =
  sqrt0(a).flatMap( b =>
  divTen0(b).flatMap( c =>
  acos0(c).map( d => d )))

scala> f30(64.0)
res28: Option[Double] = Some(0.6435011087932843)
```

```
def f3O(a:Double) =
  sqrtO(a)    .flatMap ( b =>
  divTenO(b) .flatMap ( c =>
  acosO(c)    .map      ( d =>
  d )))
```

```
def f30(a:Double) =
  sqrt0(a)    .flatMap ( b =>
  divTen0(b) .flatMap ( c =>
  acos0(c)    .map      ( d =>
  d )))

def f40(a:Double) = for {
  b <- sqrt0(a)
  c <- divTen0(b)
  d <- acos0(c)
} yield d
```

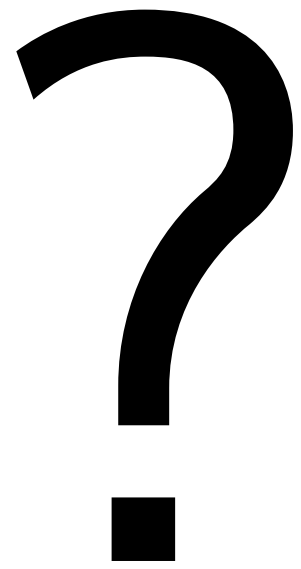
```
scala> f40(64.0)
res29: Option[Double] = Some(0.6435011087932843)
```

```
scala> f40(64.0)
res29: Option[Double] = Some(0.6435011087932843)

scala> f40(200.0)
res30: Option[Double] = None
```

```
def f4L(a:Double) = for {
  b <- sqrtL(a)
  c <- divTenL(b)
  d <- acosL(c)
} yield d
```

```
def f4L(a:Double) = for {
  b <- sqrtL(a)
  c <- divTenL(b)
  d <- acosL(c)
} yield d

scala> f4L(64.0)
res31: List[Double] = List(0.6435011087932
843, 2.498091544796509
```

```
def f4L(a:Double) = for {
  b <- sqrtL(a)
  c <- divTenL(b)
  d <- acosL(c)
} yield d

scala> f4L(64.0)
res31: List[Double] = List(0.6435011087932
843, 2.498091544796509

scala> f4L(200.0)
res32: List[Double] = List()
```



Script:

https://gist.github.com/grzegorzbalcerek/b05cdda143010e0374f5

Exercises:

https://gist.github.com/grzegorzbalcerek/d9aaa2bbf29cecdff368

Solutions:

https://gist.github.com/grzegorzbalcerek/8944fd2c86a91d429254

```scala
import language.higherKinds
trait Monad1[M[_]] {

  def id[A](a: A): M[A]

  def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B] = ???

  def join[A](mma: M[M[A]]): M[A] = ???

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] = ???

  def map[A,B](ma: M[A], f: A => B): M[B] = ???

}
```

```scala
import language.higherKinds
trait Monad2[M[_]] {

  def id[A](a: A): M[A]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

  def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C] = ???

  def join[A](mma: M[M[A]]): M[A] = ???

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] = ???

  def map[A,B](ma: M[A], f: A => B): M[B] = ???

}
```

```scala
import language.higherKinds
trait Monad3[M[_]] {

  def id[A](a: A): M[A]

  def map[A,B](ma: M[A], f: A => B): M[B]

  def join[A](mma: M[M[A]]): M[A]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B] = ???

  def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C] = ???

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] = ???

}
```

```scala
import language.higherKinds
trait Monad1[M[_]] {

  def id[A](a: A): M[A]

  def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B] =
    compose((_:Unit) => ma, f)(())

  def join[A](mma: M[M[A]]): M[A] =
    flatMap(mma, identity[M[A]])

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] =
    flatMap(ma, (a:A) =>
    flatMap(mab, (f:A=>B) =>
    id(f(a))))

  def map[A,B](ma: M[A], f: A => B): M[B] =
    ap(ma, id(f))

}
```

```scala
import language.higherKinds
trait Monad2[M[_]] {

  def id[A](a: A): M[A]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]

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

  def join[A](mma: M[M[A]]): M[A] =
    flatMap(mma, identity[M[A]])

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] =
    flatMap(ma, (a:A) =>
    flatMap(mab, (f:A=>B) =>
    id(f(a))))

  def map[A,B](ma: M[A], f: A => B): M[B] =
    ap(ma, id(f))

}
```

```
import language.higherKinds
trait Monad3[M[_]] {

  def id[A](a: A): M[A]

  def map[A,B](ma: M[A], f: A => B): M[B]

  def join[A](mma: M[M[A]]): M[A]

  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B] =
    join(map(ma,f))

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

  def ap[A,B](ma: M[A], mab: M[A => B]): M[B] =
    flatMap(ma, (a:A) =>
    flatMap(mab, (f:A=>B) =>
    id(f(a))))

}
```