Théorie des catégories pour les programmeurs

@karolchmist

Pourquoi

- Langage commun
  - Scalaz, Cats, Haskell
  - Comme les Design patterns
- Bases mathématiques solides
- For fun !

Catégories

- Objets

  a, b, c

- Morphismes

  $f : a \to b$

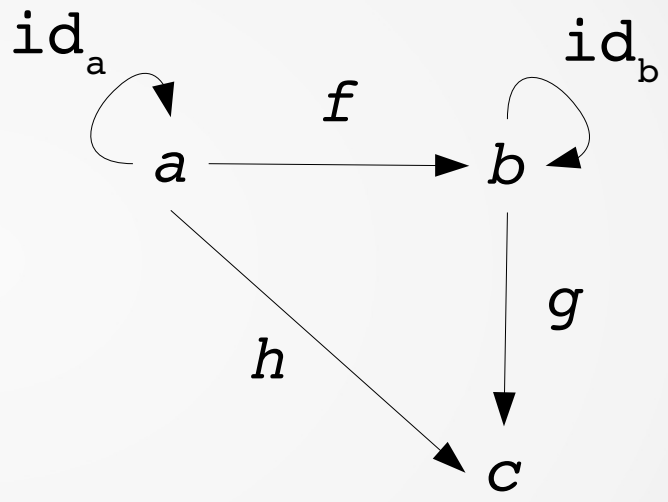
  $g : b \to c$

- Composition

  $h = g \circ f$

- Morphisme d'identité

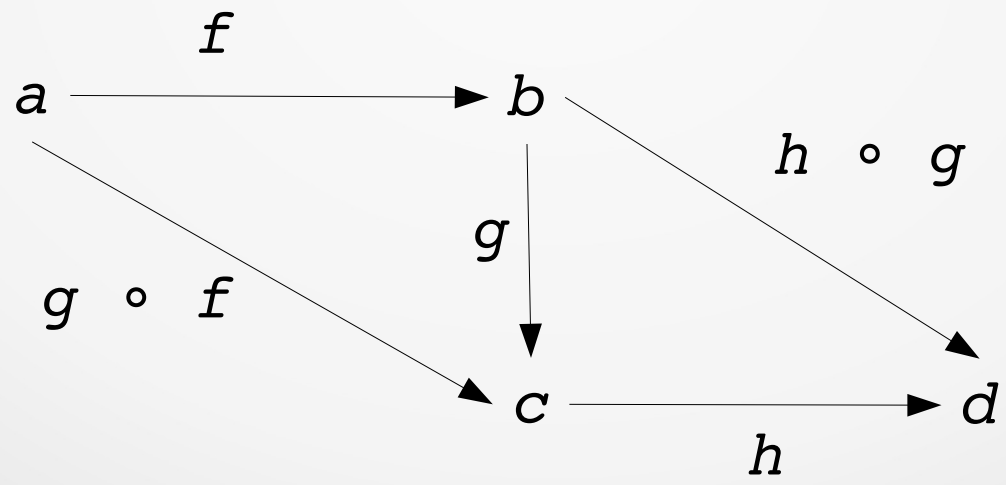
  $\mathrm{id}_b \circ f = f = f \circ \mathrm{id}_a$



Catégories

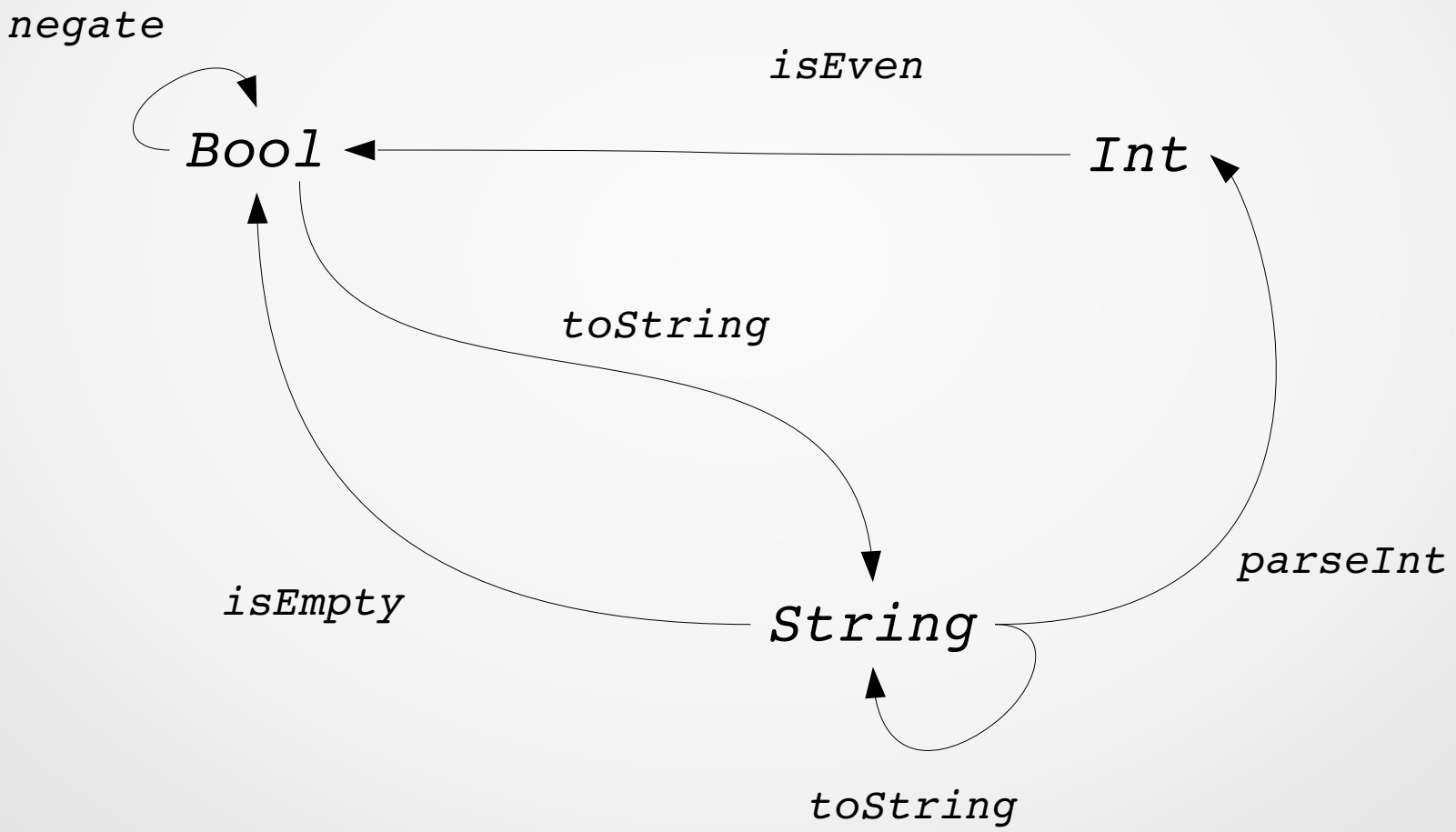
Associativité

$$h \circ (g \circ f) = h \circ g \circ f = (h \circ g) \circ f$$



Catégories

- Exemple : Système de types



Monoïdes

- Structure avec
  - opération binaire associative
  - élément neutre

Monoïdes

- Ensemble des entiers naturels muni de l'addition

  opération binaire : +

  élément neutre : 0

  0 + 3 = 3 = 3 + 0

  associativité :

  1 + (2 + 3) = (1 + 2) + 3

Monoïdes

- Ensemble des entiers naturels muni de la multiplication

  opération binaire : *

  élément neutre : 1

  1 * 5 = 5 = 5 * 1

  associativité :

  3 * (2 * 5) = (3 * 2) * 5

Monoïdes

- Les chaînes de caractères

  opération binaire : concaténation

  élément neutre : ""

  "" + "abc" = "abc" = "abc" + ""

  associativité

  ("ab" + "c") + "de" = "ab" + ("c" + "de")

Monoïdes

- Interprétation en théorie des catégories

Monoïde = Catégorie avec un objet

type = objet

opération binaire = morphismes

élément neutre = morphisme d'identité

Monoïdes

- opération binaire + élément neutre...

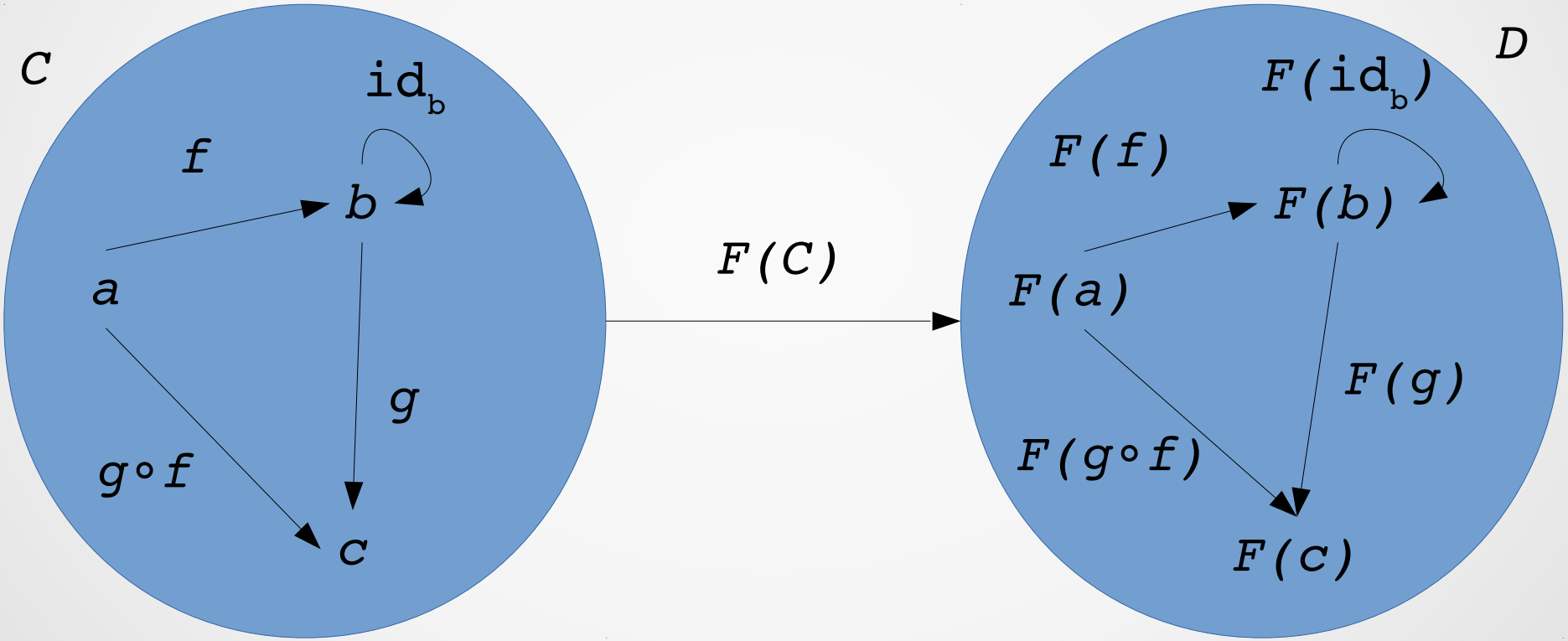
```
reduce/fold

List(2,5,1).fold(0)(_ + _)

List(3,19,8).fold(1)(_ * _)
```

Foncteurs

- Mapping entre deux catégories C et D



de tous les objects

de tous les morphismes

Foncteurs

- Loi d'identité

$$F(id_a) = id_{F(a)}$$

- Loi de composition

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

Foncteurs

- Exemples : Foncteur de morphisme d'identité

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

Some(3).map(id) == id(Some(3))

None.map(id) = id(None)
```

Foncteurs

- Exemple : Composition des foncteurs

```
case class Address(country: String)
case class Person(address: Address)
val p = Person(Address("France"))

Some(p).map{_.address.country} ==
Some(p).map{_.address}.map{_.country}
```

Foncteurs

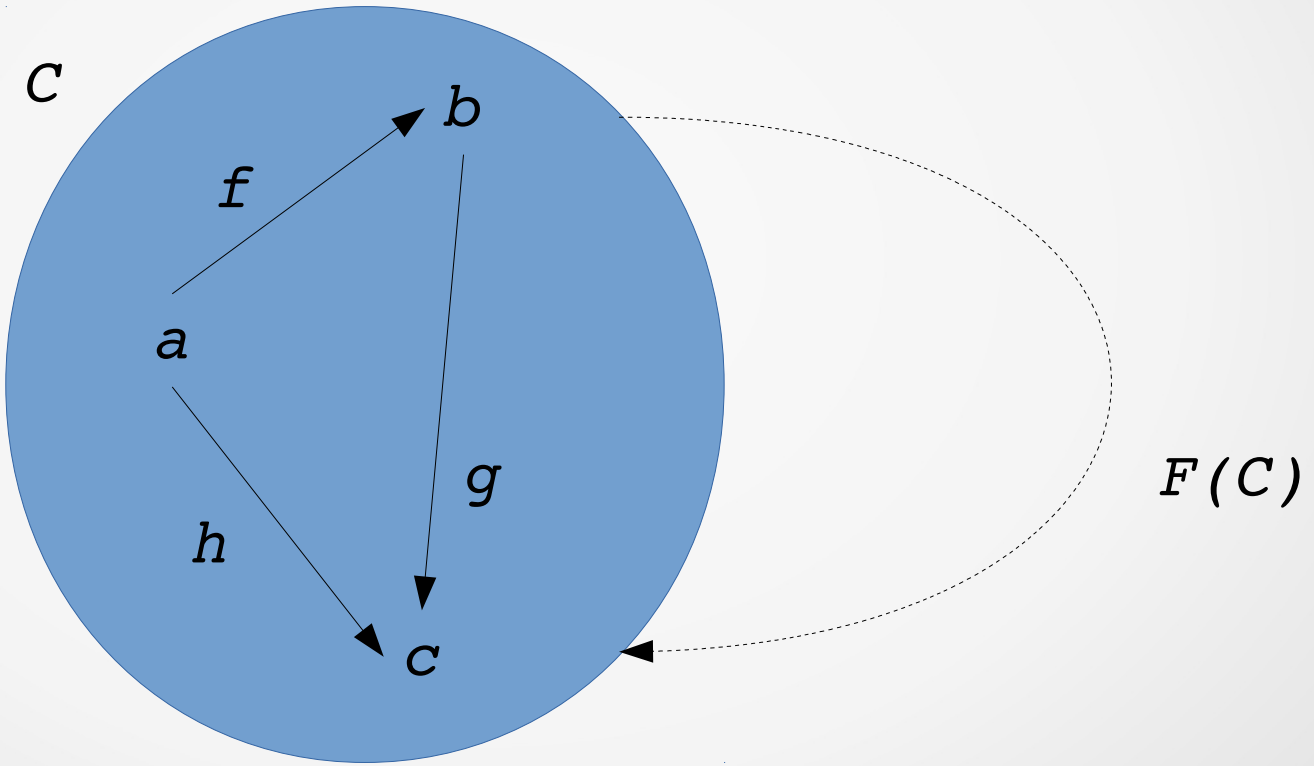
- Exemple : Mapping avec Scalaz

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

  val isEven = (x:Int) => x % 2 == 0
  val neg = (b:Boolean) => !b
  val isOdd = isEven.map(neg)
```

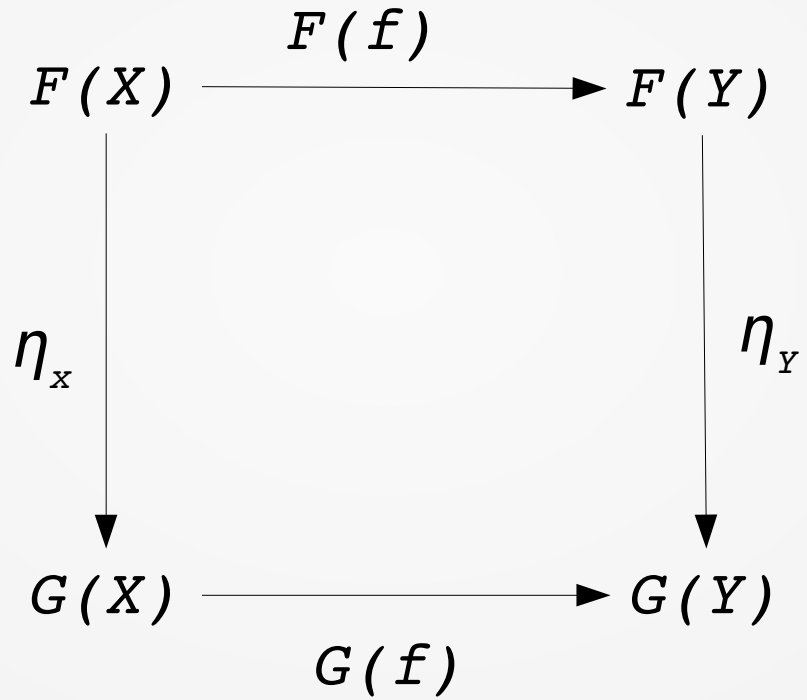
Function composition !

Endofoncteur



Transformation naturelle

- Mapping entre deux foncteurs

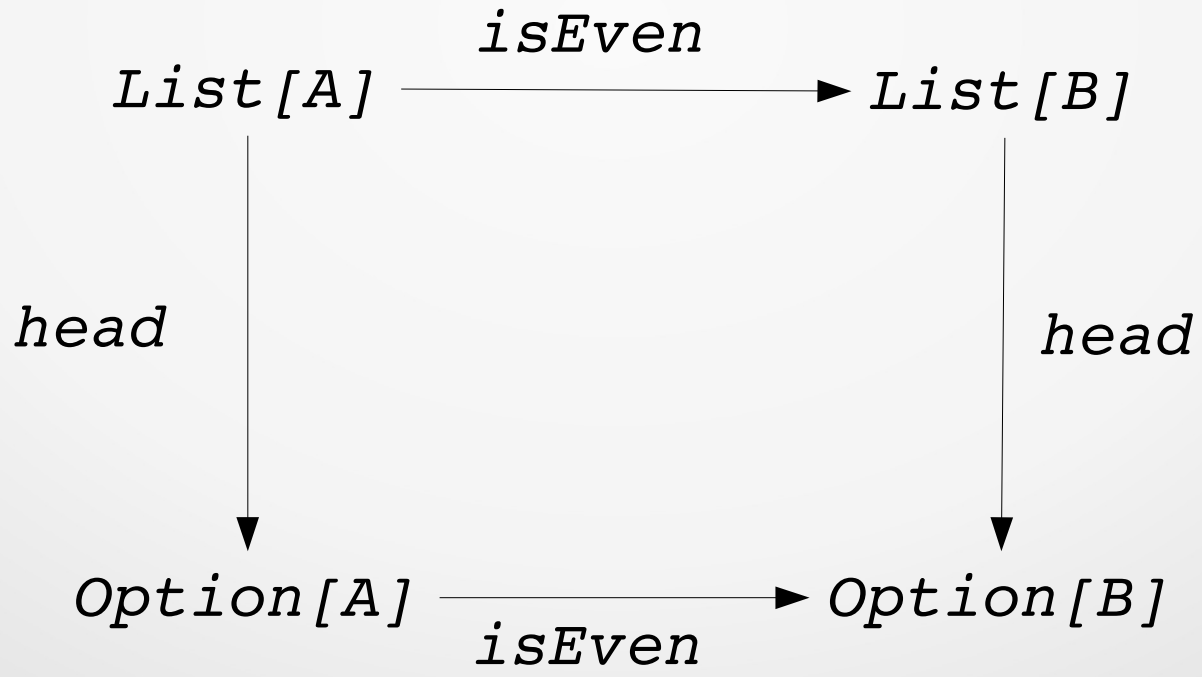


$$\eta_Y \circ F(f) = G(f) \circ \eta_X$$

Transformation naturelle

- Exemple : naturalité de head

```
def head[E](xs : List[E]) : Option[E]
def isEven(a:Int) : Bool = a % 2 == 0
```



Transformation naturelle

- Exemple : naturalité de head

```
def head[E](xs : List[E]) : Option[E]
def isEven(a:Int) : Bool = a % 2 == 0

List(20,3,49).map(isEven).head
    == Option(true)

List(20,3,49).head.map(isEven)
    == Option(true)
```

Transformation naturelle

- head
- last
- sequence
- concat

Monades


MONADE EST UN MONOÏDE DANS
LA CATÉGORIE DES ENDOFONCTEURS
QUEL EST LE PROBLÈME ?
imgflip.com

Monades

```
case class Person(id: Int, address: Address)
case class Address(street: String)


def findPersonById(id: Int) : Person = ...

def formatStreet(personId : Int) : String = {
  val p = findPersonById(personId)
  if(p != null && p.address != null && p.address.street != null) {
      return "Adress : " + p.address.street
    } else {
      return ""
    }
}
```

Monades

```
case class Person(id: Int, address: Option[Address])
case class Address(street: Option[String])

def findPersonById(id: Int) : Option[Person] = ...

def formatStreet(personId : Int) : String = {
  val p = findPersonById(personId)
  if(p.isDefined) {
    if(p.get.address.isDefined) {
      if(p.get.address.get.street.isDefined)
        return p.get.address.get.street
          .map(s => s"Address : $s")
          .getOrElse("")
    }}
  return ""
}
```

Monades

```
class Option[+A] {
  def flatMap(f: A => Option[B]): Option[B] =
    if (isEmpty) None
    else f(this.get)

  def map(f: A => B): Option[B] =
    if (isEmpty) None
    else Some(f(this.get))
}
```

Monades

```
case class Person(id: Int, address: Option[Address])
case class Address(street: Option[String])

def findPersonById(id: Int): Option[Person] = ???

def getStreetByPersonId(personId: Int): Option[String] =
  findPersonById(personId).flatMap {
    (p:Person) => p.address.flatMap {
      a => a.street.map {
        str => s"Address : $str"
      }
    }
  }
```

Monades

```
case class Person(id: Int, address: Option[Address])
case class Address(street: Option[String])

def getStreet(personId: Int) : Option[String] =
  for {
    person <- findPersonById(personId)
    address <- person.address
    street <- address.street
 } yield s"Adress : $street"
```

Monades

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

Monades

```
def associativityLaw[A, B, C]
   (a: Monad[A])
   (f: A => Monad[B],
    g: B => Monad[C]) =
flatMap(flatMap(a)(f))(g)
== flatMap(a)(v => flatMap(f(v))(g))
```

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Monades

```
def leftUnitLaw[A,B](a:A)(f: A=>Monad[B]) =
    flatMap(unit(a))(f) == f(a)

def rightUnitLaw[A,B](a:Monad[A]) =
    flatMap(a)(unit) == a
```

$$id_b \circ f = f \circ id_a$$

Monades

- Option
- List
- Either
- Future
- IO
- Writer
- Reader
- State
- ...

Conclusion

On utilise déjà
la théorie des catégories !

Sources

Paul Chiusano, Rúnar Bjarnason : Functional Programming in Scala

Juan Pedro Villa Isaza : Category Theory Applied to Functional Programming

Bartosz Milewski : Category Theory for Programmers

Eugene Yokota : Learning Scalaz

Questions ?

Merci !

@karolchmist