
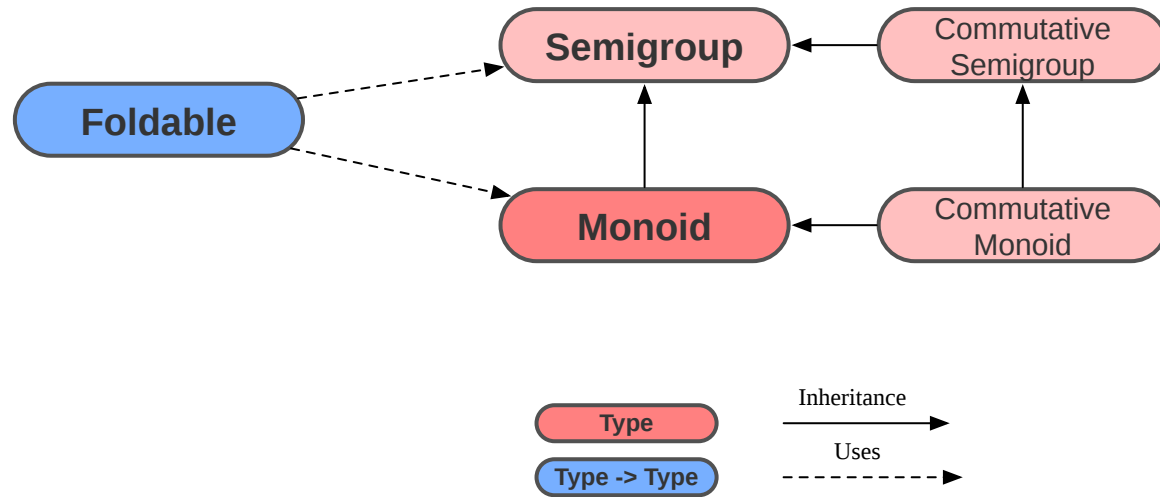


# FOUNDATION

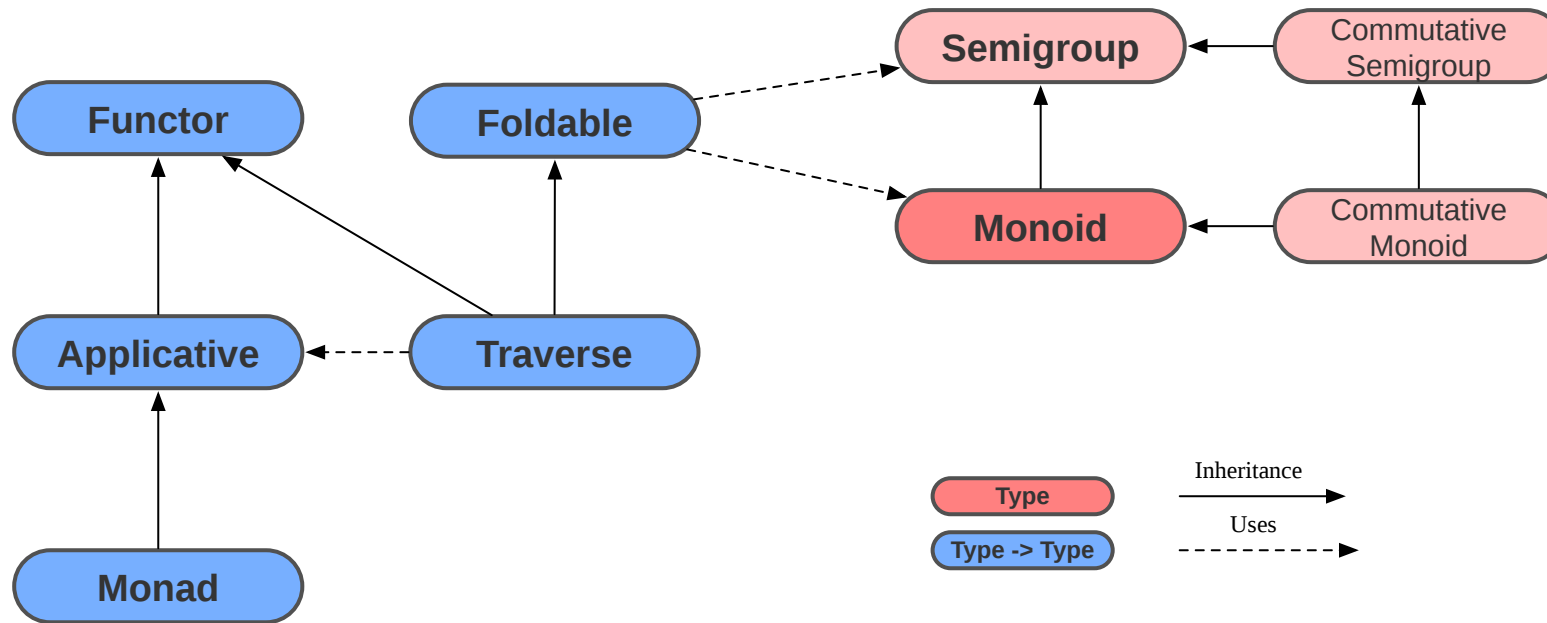


Functors

# Previously



# Plan



# Why Functors? Where are they used?



# Where are Functors used?

Data structures: List, Vector, Map, Set, Graph

Error handling: Option, Either, Validated

Async: IO, Future

Parsing, Serialisation, Configuration, etc ...



# Plan

- Functor hierarchy
- 20+ generic functions
- Variance
- Typeclass coherence



# Functor



# Functor

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





# Functor

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

## Functor is a higher kinded typeclass

- It applies to **type constructor** with a **single** hole
- But **types** like Int, String **cannot** be a Functor

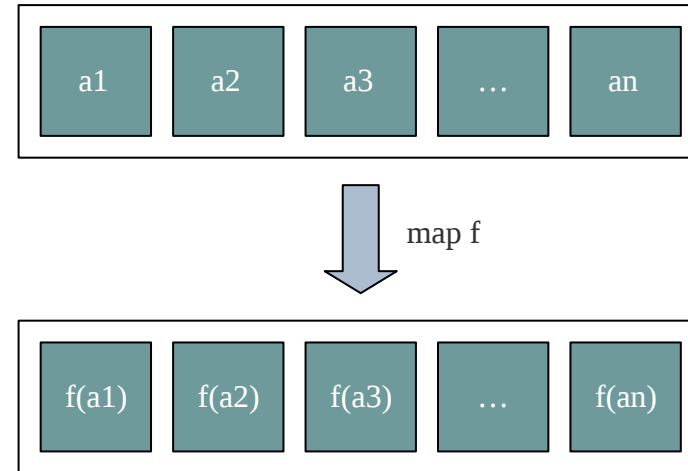


Which  $F[_]$  is a Functor?



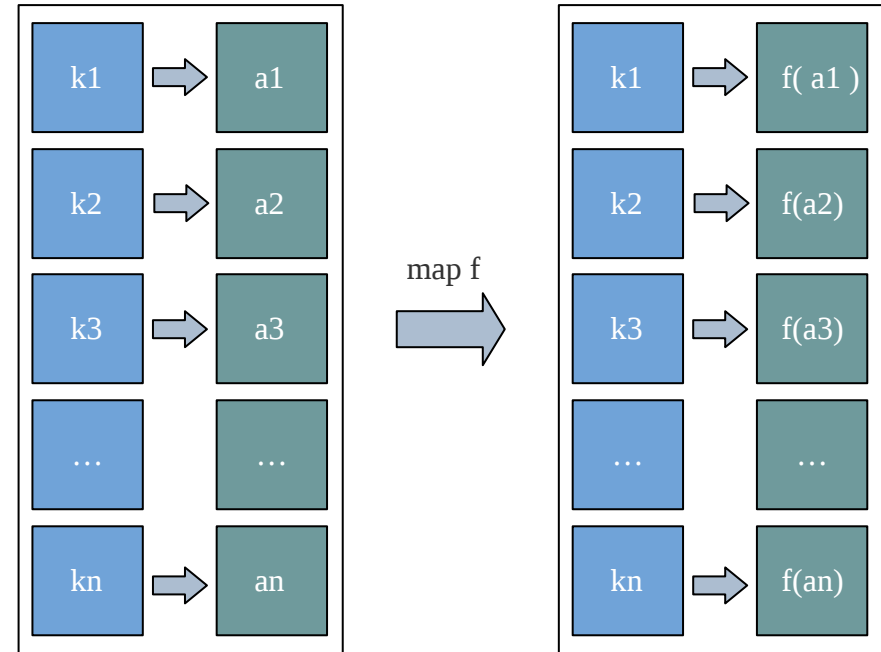
# F is a container

- List, Vector, Stream
- Option, Try



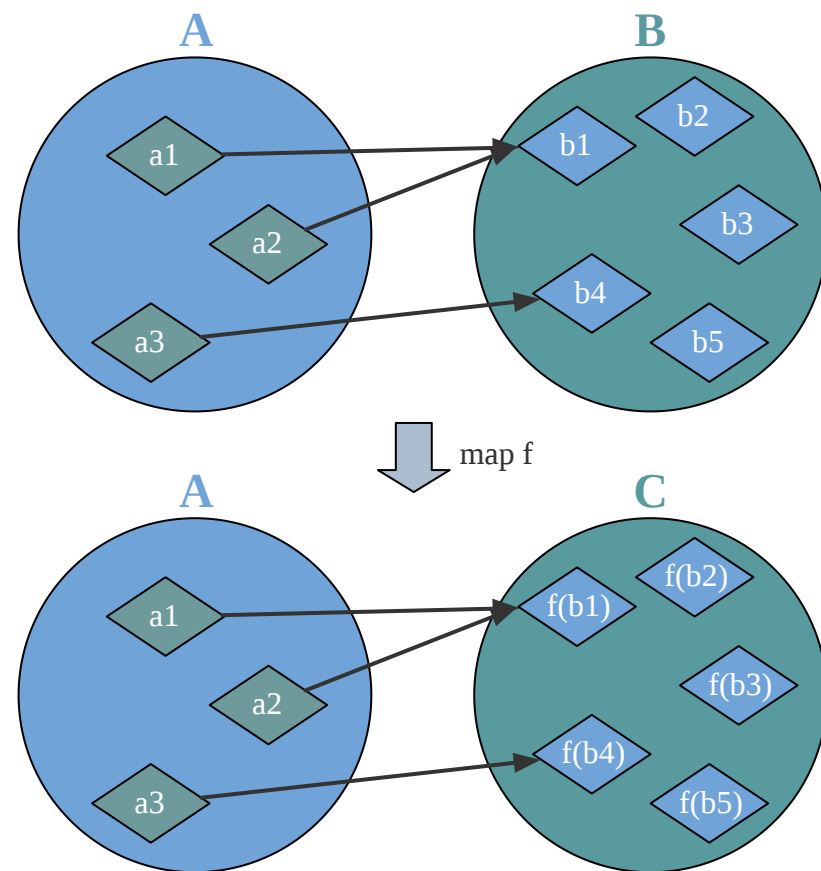
# F is a container

- List, Vector, Stream
- Option, Try
- Map[K, ?], Either[E, ?]

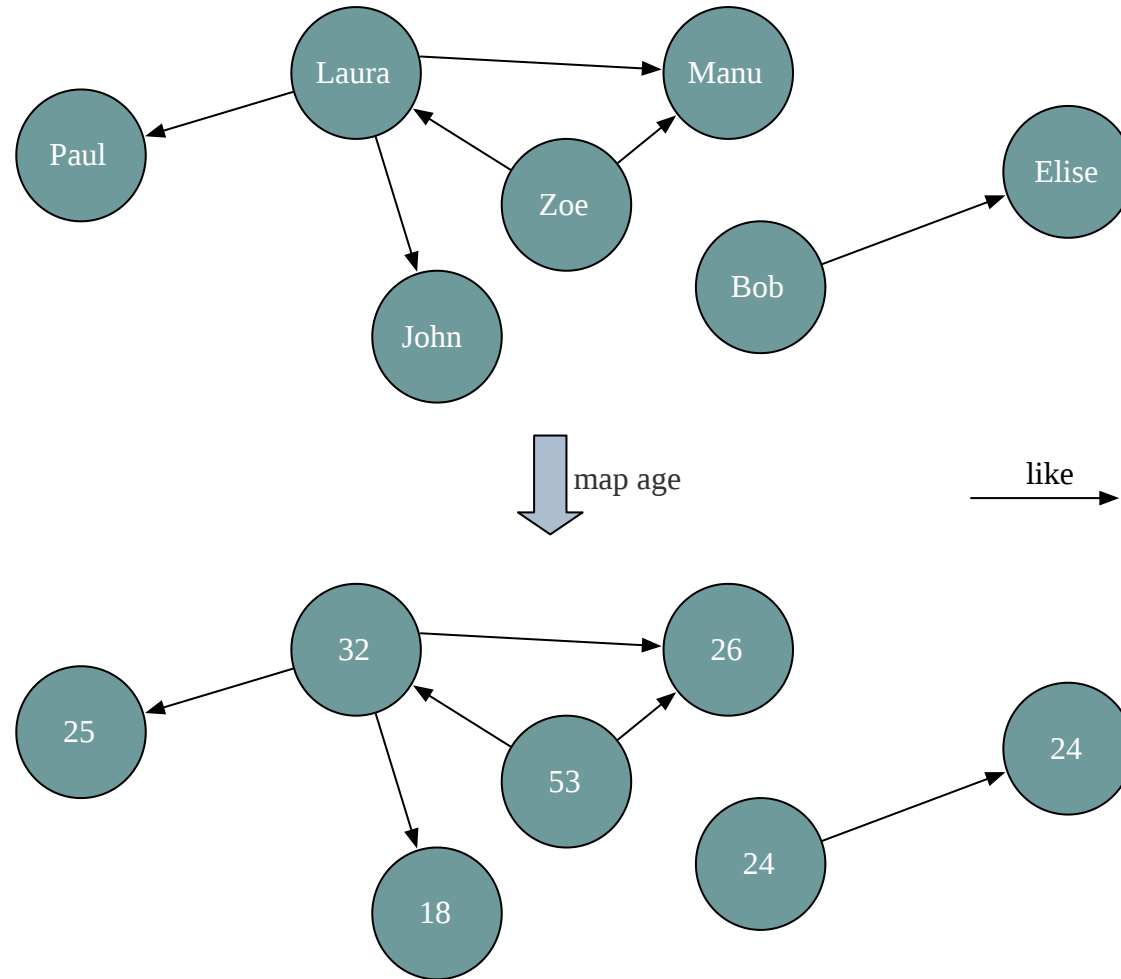


# F is a container

- List, Vector, Stream
- Option, Try
- Map[K, ?], Either[E, ?]
- $A \Rightarrow ?$



# A Functor does not alter the structure



# F is an effect

```
type Failure[A] = Option[A] or Either[String, A]
type Nondeterminism[A] = List[A] or Vector[A]
type SideEffect[A] = IO[A]
type MutableState[A] = State[Int, A]
type ImmutableState[A] = Reader[Int, A]
type Console[A] = ...
type Logger[A] = ...
```

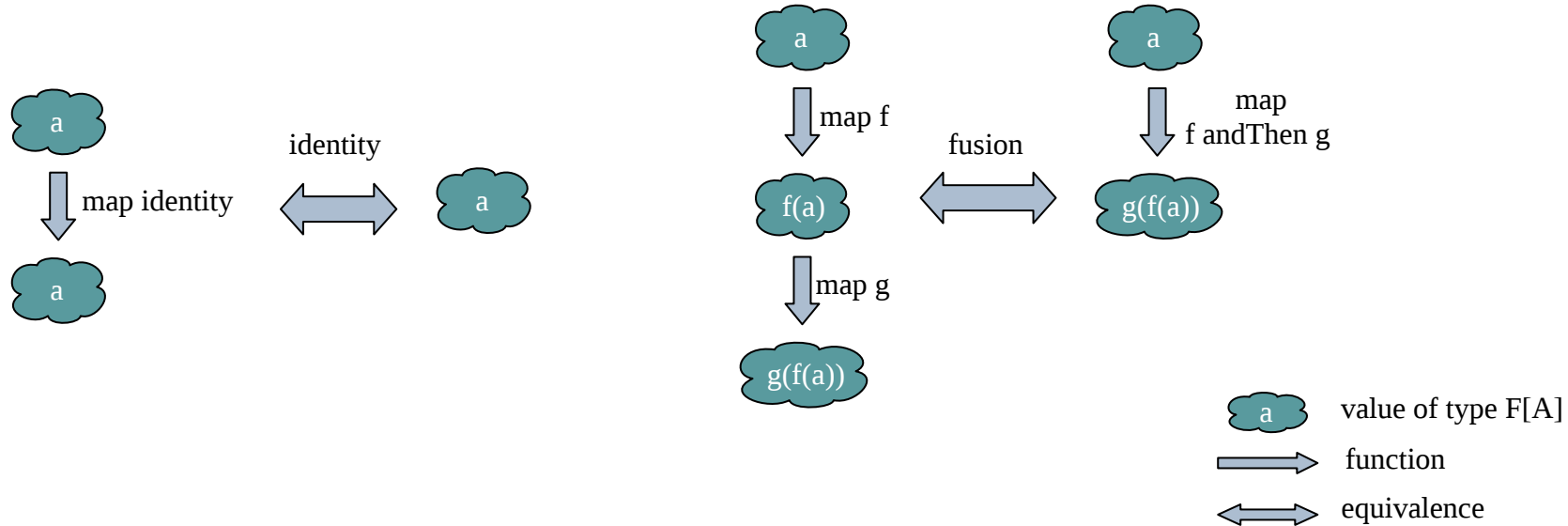
A Functor is an abstraction to update an effectful value **without** altering its effect



# Functor Laws

```
forall(fa: F[A] => fa.map(identity) == fa)
```

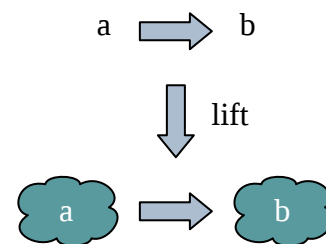
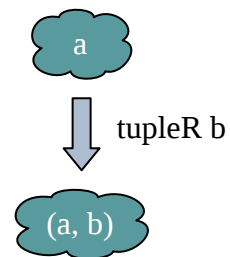
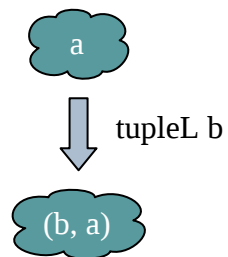
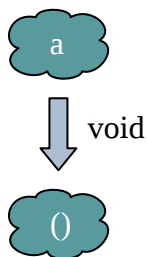
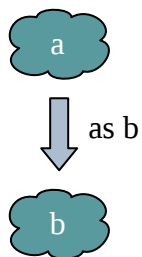
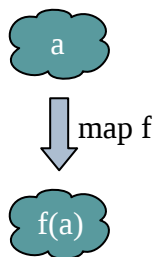
```
forall((fa: F[A], f: A => B, g: B => C) =>  
  fa.map(f).map(g) == fa.map(f andThen g)  
)
```



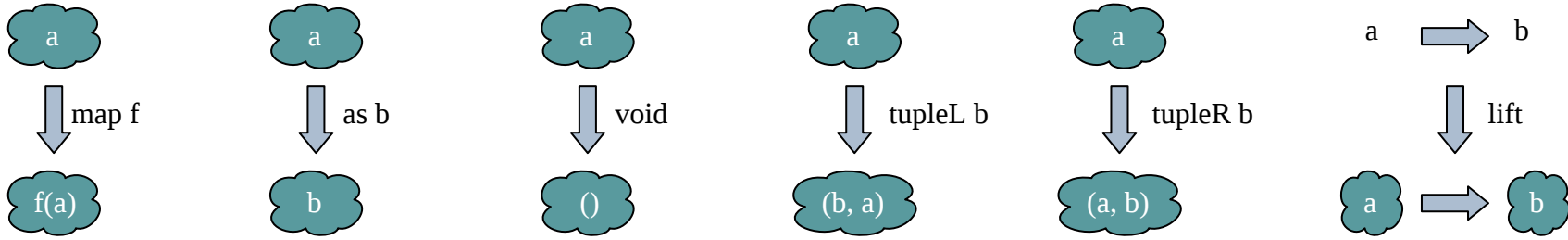


# Functor API

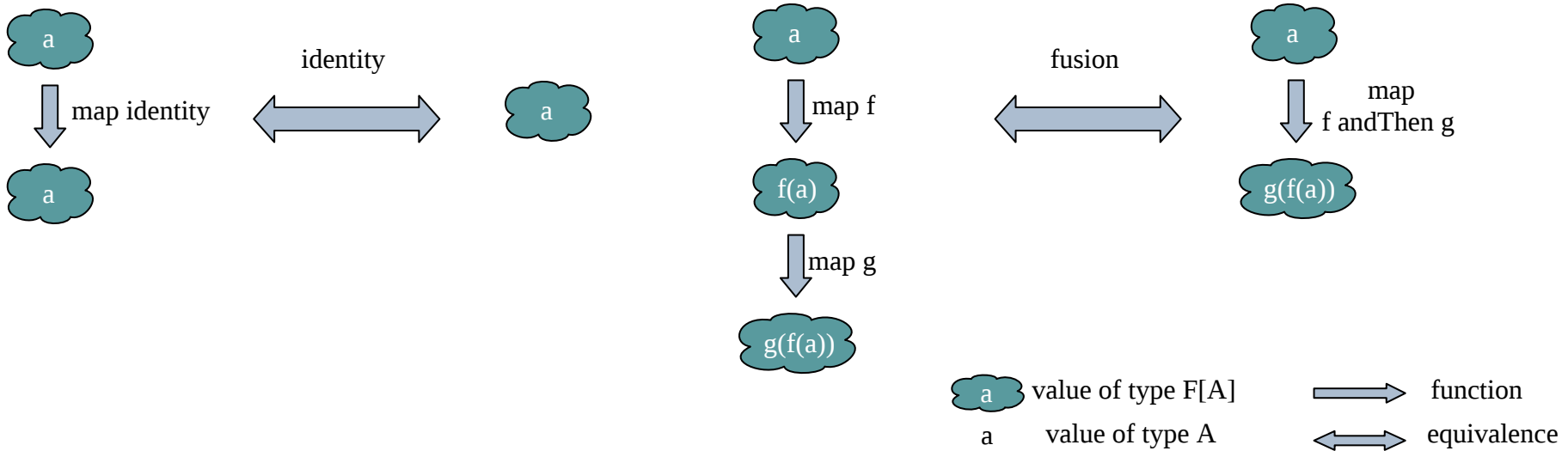
```
trait Functor[F[_]] {  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
  
  def as[A, B](fa: F[A])(value: B): F[B]  
  def void[A] (fa: F[A]): F[Unit]  
  
  def tupleL[A, B](fa: F[A])(value: B): F[(B, A)]  
  def tupleR[A, B](fa: F[A])(value: B): F[(A, B)]  
  
  def lift[A, B](f: A => B): F[A] => F[B]  
}
```



# API



# Laws



# Exercise 1



# Functor

## □ ADT

```
case class Foo[A](i: Int, a: A)

sealed trait Bar[A]
case class Bar1[A](i: Int, a: A) extends Bar[A]
case class Bar2[A](b: Boolean, a: A) extends Bar[A]
case class Bar3[A](s: String) extends Bar[A]
```

## □ Function result

```
case class Producer[A](func: Int => A)
```



# Functor

## □ ADT

```
case class Foo[A](i: Int, a: A)

sealed trait Bar[A]
case class Bar1[A](i: Int, a: A) extends Bar[A]
case class Bar2[A](b: Boolean, a: A) extends Bar[A]
case class Bar3[A](s: String) extends Bar[A]
```

## □ Function input

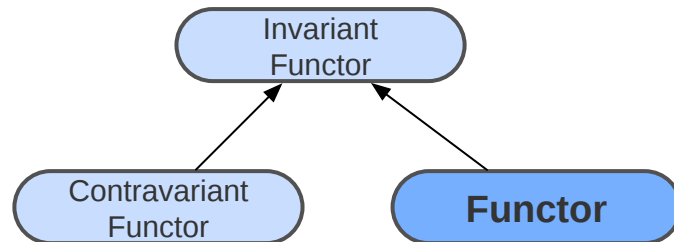
```
case class Predicate[A](func: A => Boolean)
```

## □ Function result

```
case class Producer[A](func: Int => A)
```



# Functor variance



```
trait InvariantFunctor[F[_]]{  
  def imap[A, B](fa: F[A])(f: A => B)(g: B => A): F[B]  
}  
  
trait ContravariantFunctor[F[_]] extends InvariantFunctor[F] {  
  def contramap[A, B](fa: F[A])(f: B => A): F[B]  
}  
  
trait Functor[F[_]] extends InvariantFunctor[F] { // CovariantFunctor  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
}
```



# Functor variance

```
trait JsonDecoder[A]{  
  def decode(value: Json): A  
}  
  
trait JsonEncoder[A]{  
  def encode(value: A): Json  
}  
  
trait JsonCodec[A] extends JsonDecoder[A] with JsonEncoder[A]
```

What kind of Functor is JsonDecoder, JsonEncoder and JsonCodec?



# What kind of Functor is JsonDecoder?

```
trait JsonDecoder[A]{  
  def decode(value: Json): A  
}
```





# What kind of Functor is JsonDecoder?

```
trait JsonDecoder[A]{  
  def decode(value: Json): A  
}
```

is equivalent to

```
case class JsonDecoder[A](decode: Json => A)
```

hence

```
implicit val functor: Functor[JsonDecoder] = ???
```



# What kind of Functor is JsonEncoder?

```
trait JsonEncoder[A]{  
  def encode(value: A): Json  
}
```



# What kind of Functor is JsonEncoder?

```
trait JsonEncoder[A]{  
  def encode(value: A): Json  
}
```

is equivalent to

```
case class JsonEncoder[A](encode: A => Json)
```

hence

```
implicit val functor: ContravariantFunctor[JsonEncoder] = ???
```



# What kind of Functor is JsonCodec?

```
trait JsonCodec[A] extends JsonDecoder[A] with JsonEncoder[A]
```



# What kind of Functor is JsonCodec?

```
trait JsonCodec[A] extends JsonDecoder[A] with JsonEncoder[A]
```

is equivalent to

```
case class JsonCodec[A](  
  decode: Json => A,  
  encode: A => Json  
)
```

hence

```
implicit val functor: InvariantFunctor[JsonCodec] = ???
```



# Variance

Type	Example	A	B
Product	(A, B)	Covariant	Covariant
Sum	Either[A, B]	Covariant	Covariant
Function	$A \Rightarrow B$	Contravariant	Covariant
Endo Function	$A \Rightarrow A$	Invariant	N/A



# Variance

Type	A	B	C
$(A \Rightarrow B) \Rightarrow C$	Covariant	Contravariant	Covariant

[Thinking with types](#) by Sandy Maguire



# Generalising Functor

```
Functor          = FunctorOf (->) (->)  
ContravariantFunctor = FunctorOf (<-) (->)  
InvariantFunctor  = FunctorOf Iso Iso  
Bifunctor  
Profunctor  
...
```

See [Functor-Of](#) from Vladimir Ciobanu and @Iceland\_jack





A Functor is an abstraction to update an effectful value  
without altering its effect



What if we have more than 1 effect?



How can we combine effects? e.g.  $F[A]$  and  $F[B]$

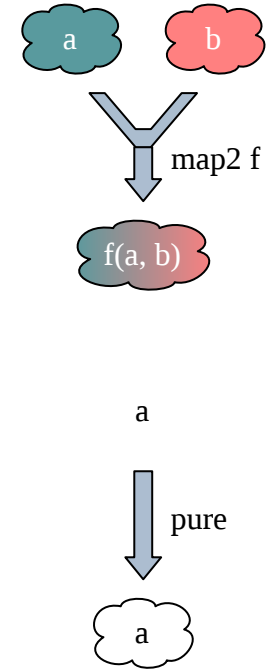


# Applicative



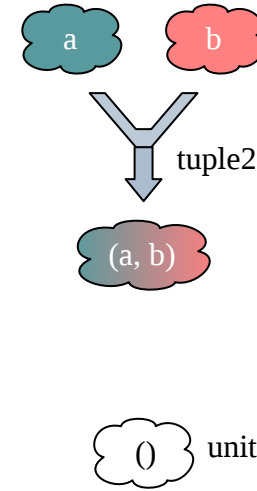
# Applicative

```
trait Applicative[F[_]] extends Functor[F] {  
  
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]  
  
  def pure[A](a: A): F[A]  
  
}
```



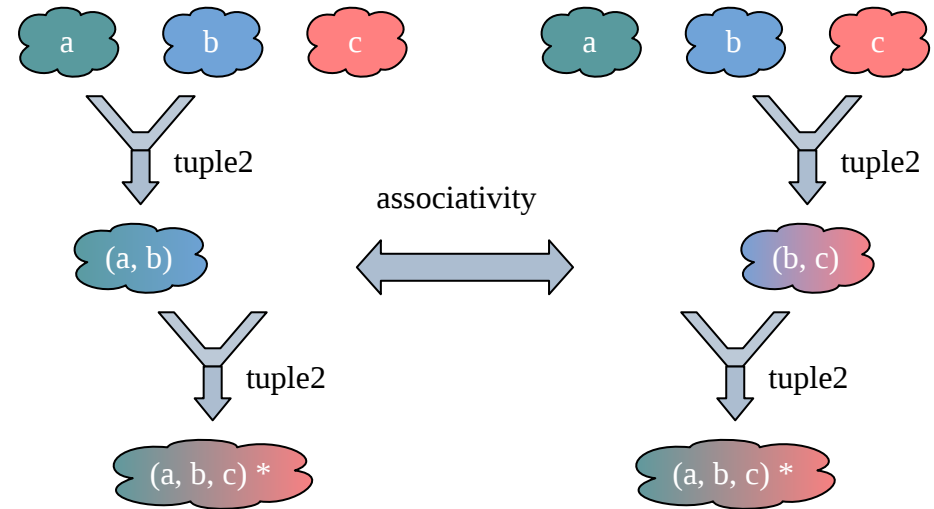
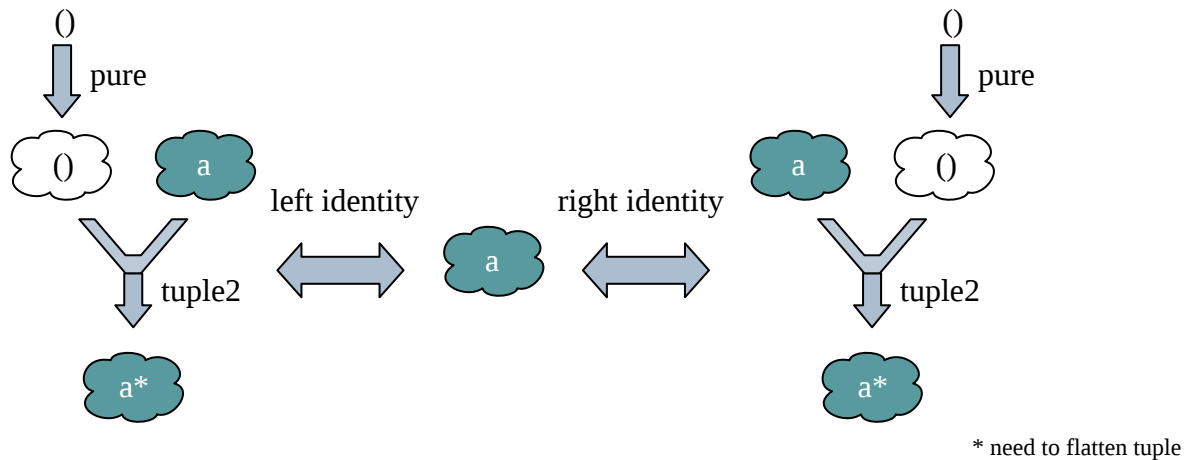
# Applicative alternative encoding

```
trait Applicative[F[_]] extends Functor[F] {  
  
  def tuple2[A, B](fa: F[A], fb: F[B]): F[(A, B)]  
  
  def unit: F[Unit]  
  
}
```



# Applicative laws

```
forall(fa: F[A] => tuple2(pure(()), fa )    == fa)    // equalities hold if we flatten tuples,  
forall(fa: F[A] => tuple2(fa                , pure(())) == fa)    // e.g. (a, ()) == a == ((), a)  
  
forall((fa: F[A], fb: F[A], fc: F[C]) =>  
  tuple2(tuple2(fa, fb), fc) == tuple2(fa, tuple2(fb, fc)) // ((a, b), c) == (a, b, c) == (a, (b, c))  
)
```



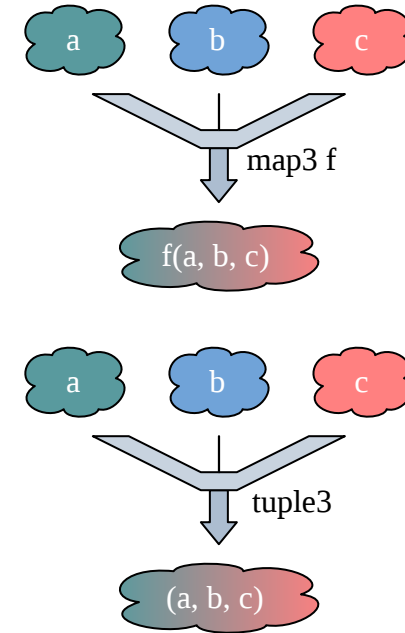
An Applicative combines 2 effects associatively  
and lift pure values into NOOP effect





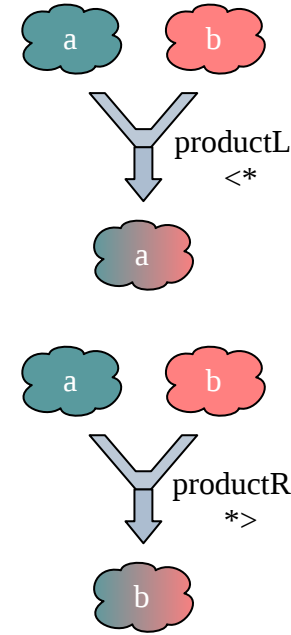
# Applicative API

```
trait Applicative[F[_]] extends Functor[F] {  
  def pure[A](a: A): F[A]  
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]  
  
  def map3[A, B, C, D](fa: F[A], fb: F[B], fc: F[C])  
    (f: (A, B, C) => D): F[D]  
  
  def tuple3[A, B, C](fa: F[A], fb: F[B], fc: F[C]): F[(A, B, C)]  
}
```

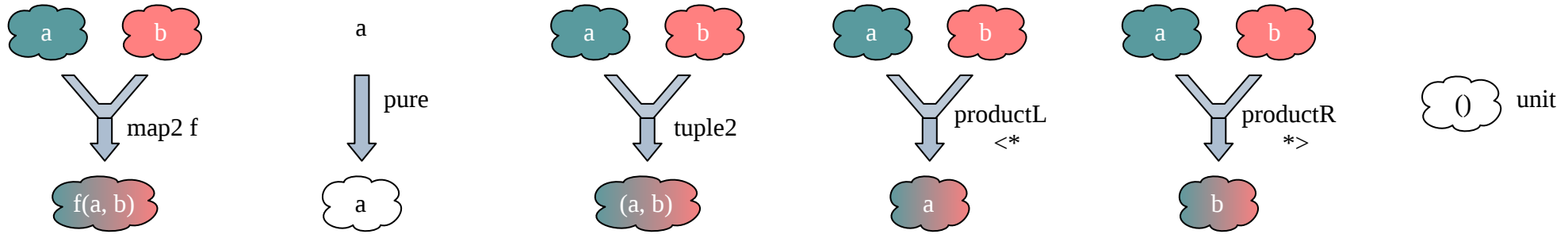


# Applicative API

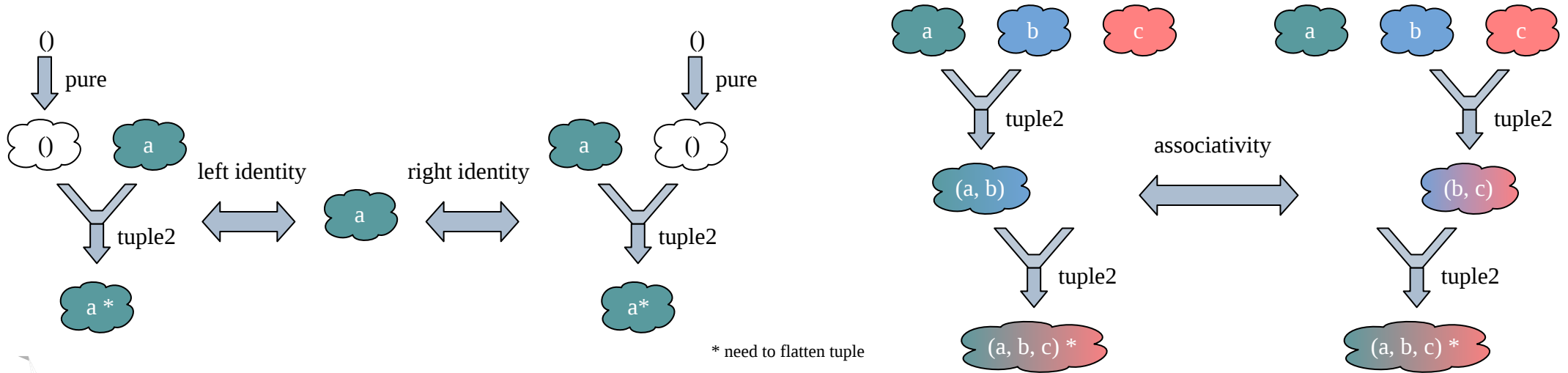
```
trait Applicative[F[_]] extends Functor[F] {  
  def pure[A](a: A): F[A]  
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]  
  
  // alias to <*  
  def productL[A, B](fa: F[A], fb: F[B]): F[A]  
  
  // alias to *>  
  def productR[A, B](fa: F[A], fb: F[B]): F[B]  
}
```



## API



## Laws



# Applicative combine effects: Failure

```
def right[A, E](a: A): Either[E, A] = Right(a)
def left [A, E](e: E): Either[E, A] = Left(e)
```

```
scala> (right(1), right("hello")).tuple2
res0: Either[Nothing,(Int, String)] = Right((1,hello))
```

```
scala> (right(1), left("an error")).tuple2
res1: Either[String,(Int, Nothing)] = Left(an error)
```

```
scala> (left("oops"), right(2)).tuple2
res2: Either[String,(Nothing, Int)] = Left(oops)
```

```
scala> (left("oops"), left("an error")).tuple2
res3: Either[String,(Nothing, Nothing)] = Left(oops)
```



# Applicative combine effects: Failure

```
scala> println((validNel(1), validNel("hello")).tuple2)  
Valid((1,hello))
```

```
scala> println((validNel(1), invalidNel("an error")).tuple2)  
Invalid(NonEmptyList(an error))
```

```
scala> println((invalidNel("oops"), validNel(2)).tuple2)  
Invalid(NonEmptyList(oops))
```

```
scala> println((invalidNel("oops"), invalidNel("an error")).tuple2)  
Invalid(NonEmptyList(oops, an error))
```



# Applicative combine effects: Nondeterminism

```
scala> (List(1,2,3), List("foo", "bar")).tuple2  
res8: List[(Int, String)] = List((1,foo), (1,bar), (2,foo), (2,bar), (3,foo), (3,bar))
```

```
scala> (List(1,2,3), Nil).tuple2  
res9: List[(Int, Nothing)] = List()
```



# Applicative combine effects: Dependency injection

```
case class State(userName: String, useLargeList: Boolean)

def greet(state: State): String = s"Welcome ${state.userName}"

def listSize(state: State): Int = if(state.useLargeList) 100 else 10

val combined: State => (String, Int) = (greet _, listSize _).tuple2
```

```
scala> combined(State("John Doe", true))
res10: (String, Int) = (Welcome John Doe,100)

scala> combined(State("Lena Doe", false))
res11: (String, Int) = (Welcome Lena Doe,10)
```



# Applicative combine effects: Side Effect

```
import cats.effect.IO  
  
val combined = (IO(println("hello")), IO(println("I love FP!"))).tuple2
```

```
scala> combined.unsafeRunSync  
hello  
I love FP!  
res12: (Unit, Unit) = ((),())
```





# Exercise 2

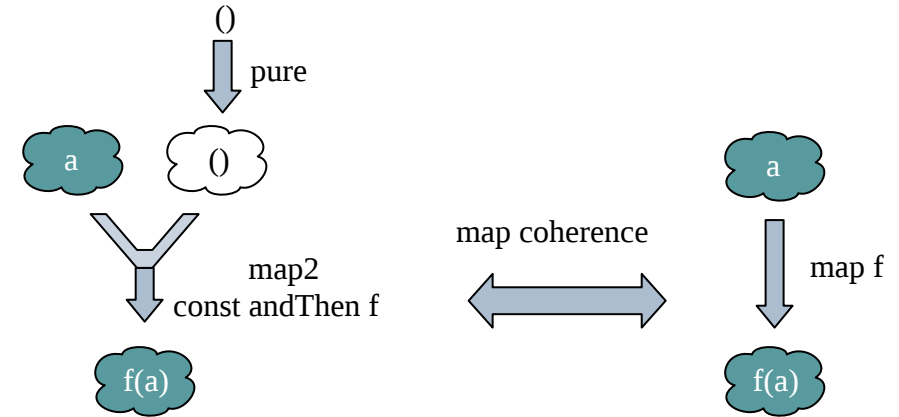


# Applicative is a Functor

```
trait Applicative[F[_]] extends Functor[F] {  
  def pure[A](a: A): F[A]  
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]  
}
```

## Coherence Law

```
forAll((fa: F[A], f: A => B) =>  
  (fa, pure(())).map2((a, _) => f(a)) == fa.map(f)  
)
```



# Applicative is a Monoidal Functor

```
combine  :: A    => A    => A
tuple2   :: F[A] => F[B] => F[(A, B)]
```

```
empty :: A
pure  :: A => F[A]
```

## Laws

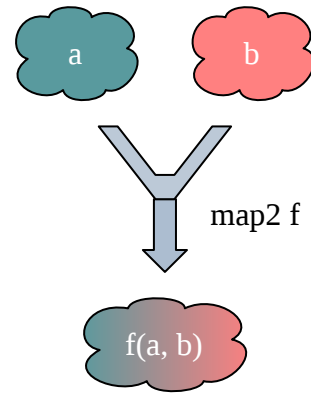
```
forall(a: A    => combine(empty, a ) == a )
forall(fa: F[A] => tuple2(unit  , fa) == fa) // equalities hold if we flatten tuples

forall(a: A    => combine(a, empty) == a )
forall(fa: F[A] => tuple2(fa, unit ) == fa)

forall(( a: A    , b: B    , c: C)    => combine(combine(a, b), c) == combine(a, combine(b, c)))
forall((fa: F[A], fb: F[A], fc: F[C]) => tuple2(tuple2(fa, fb), fc) == tuple2(fa, tuple2(fb, fc)))
```



# Applicative combines effects

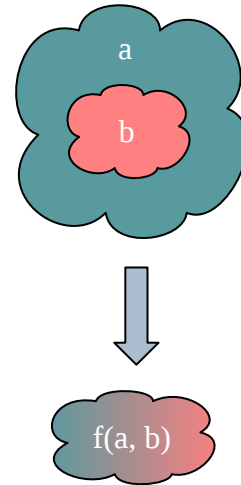


What if one effect depends on another one?

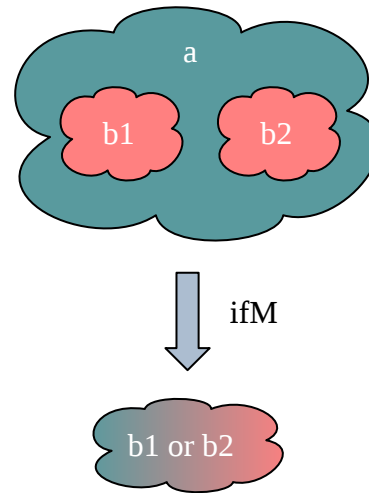
e.g.  $F[B]$  depends on  $A$



# Nested effects

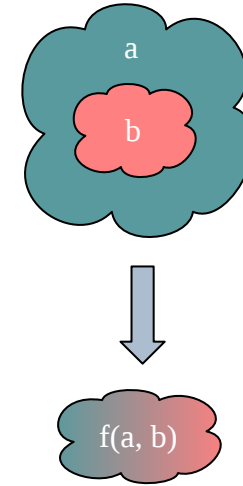


# Nested effects: ifM



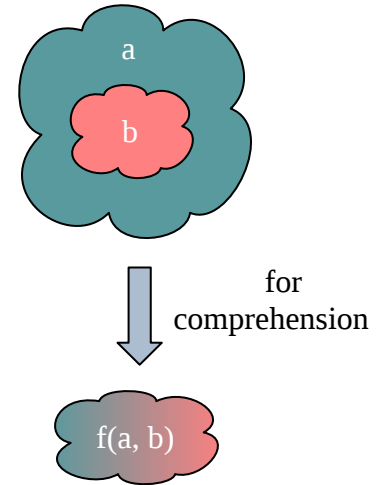
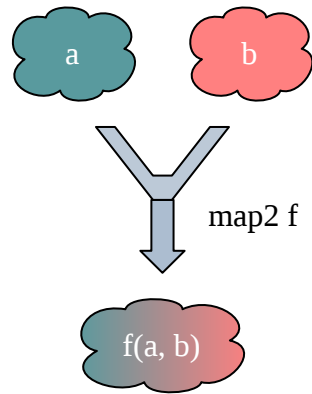
# Nested effects

```
for {  
  a <- fa  
  b <- fb(a)  
} yield f(a, b)
```





# Independent vs dependent effects



# For comprehension

```
val res: F[D] =  
  for {  
    a: A <- fa: F[A]  
    b: B <- fb: F[B]  
    c: C <- fb: F[C]  
  } yield f(a,b,c): D
```



# For comprehension

```
val res: F[D] =  
  for {  
    a: A <- fa: F[A]  
    b: B <- fb: F[B]  
    c: C <- fb: F[C]  
  } yield f(a,b,c): D
```

## Using Monad

```
val res =  
  fa.flatMap(a =>  
    fb.flatMap(b =>  
      fc.flatMap(c =>  
        f(a,b,c).pure[F]  
      )))
```

## Using FlatMap

```
val res =  
  fa.flatMap(a =>  
    fb.flatMap(b =>  
      fc.map(c =>  
        f(a,b,c)  
      )))
```



# For comprehension is for a single Monad

```
for {  
  a: A <- fa: F[A]  
  b: B <- fb: G[B] □  
  c: C <- fb: H[C] □  
} yield f(a,b,c)
```

```
scala> for {  
  |   i <- List(1,2,3)  
  |   e <- if(i == 1) Right("One") else Left("Expected 1")  
  | } yield i  
    e <- if(i == 1) Right("One") else Left("Expected 1")  
          ^
```

On line 4: error: **type mismatch**;  
found : scala.util.Either[String,Int]  
required: scala.collection.IterableOnce[?]

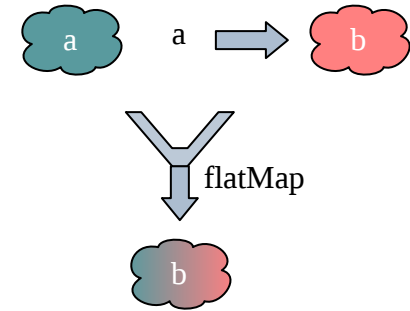


# Monad



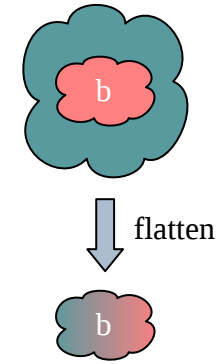
# Monad

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



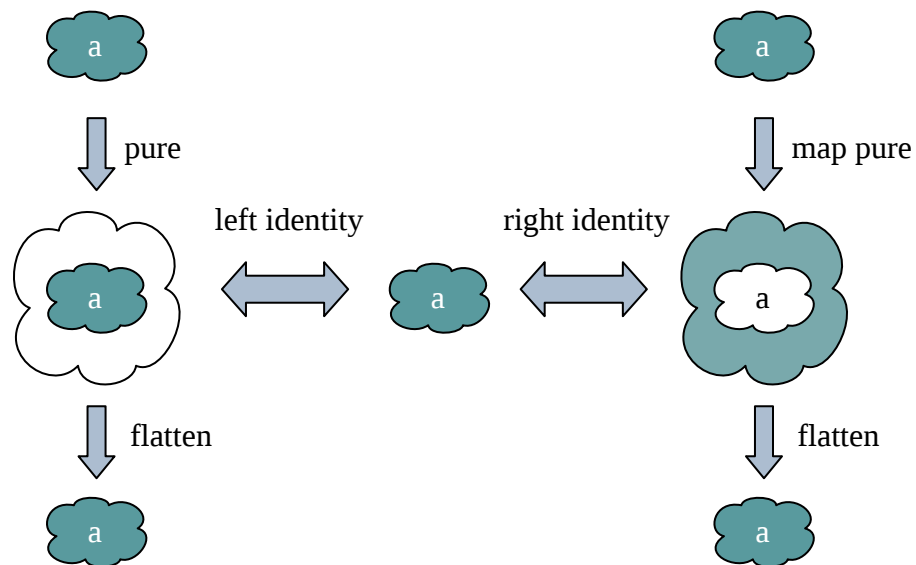
# Monad alternative encoding

```
trait Monad[F[_]] extends Applicative[F] {  
  def flatten[A](ffa: F[F[A]]): F[A]  
}
```



# Monad identity laws

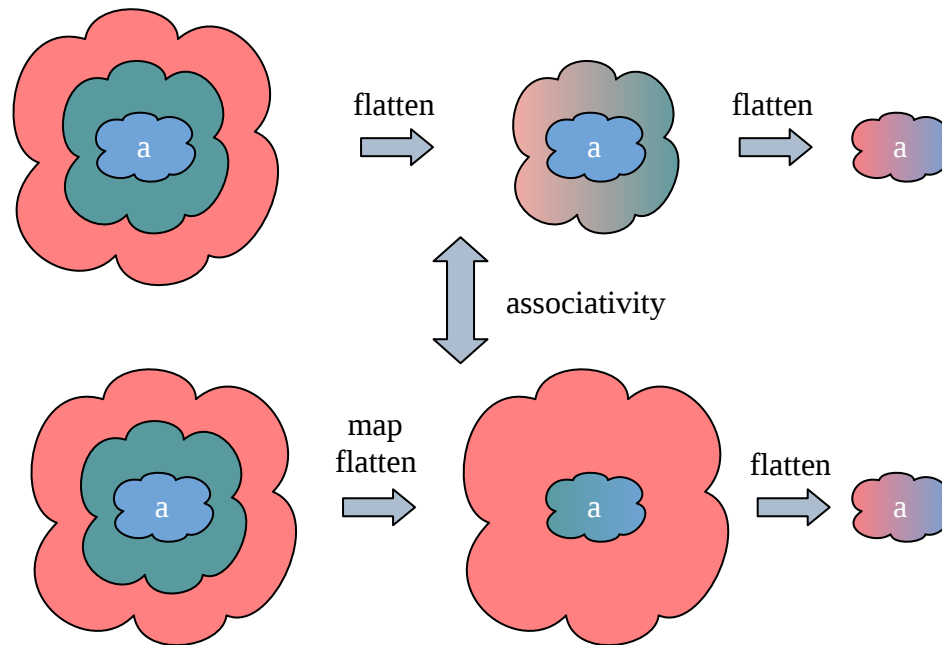
```
forall (fa: F[A] => fa.pure[F].flatten == fa)
forall (fa: F[A] => fa.map(_ . pure[F]).flatten == fa)
```





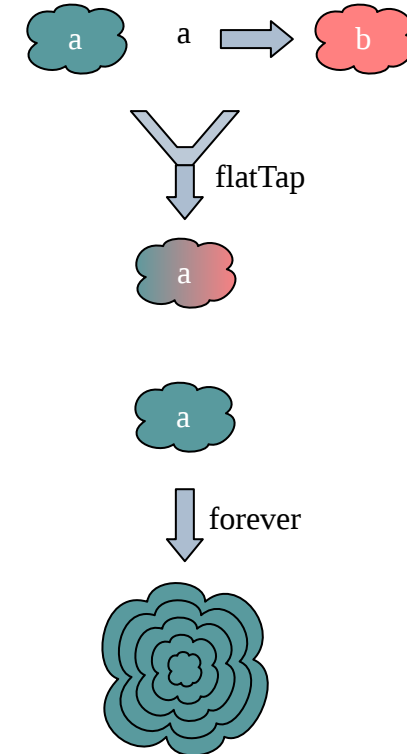
# Monad associativity law

```
forAll(fffa: F[F[F[A]]) =>  
  fa.flatten.flatten == fa.map(_.flatten).flatten  
)
```

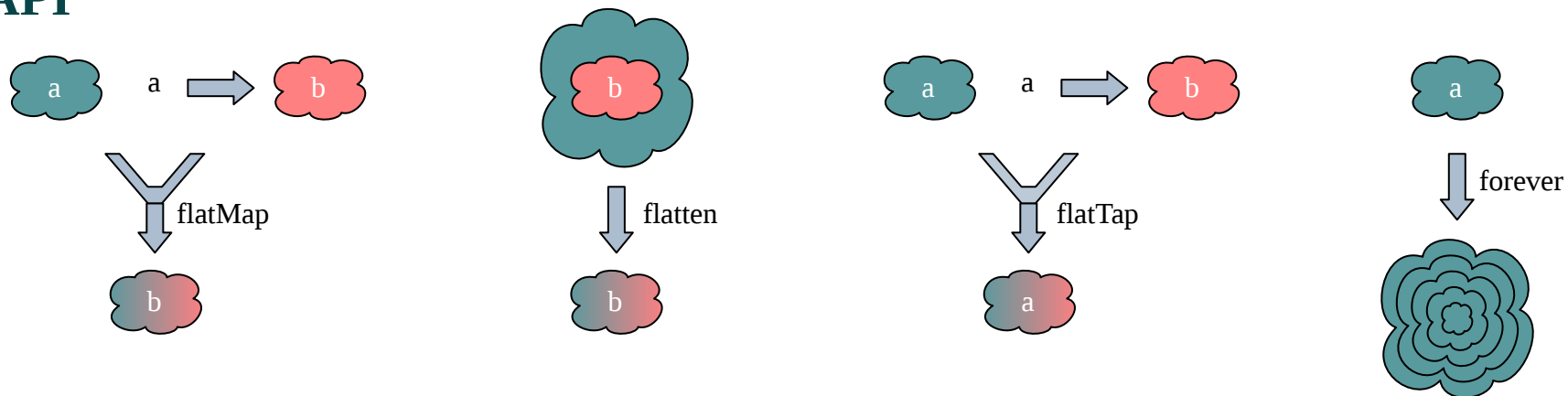


# Monad API

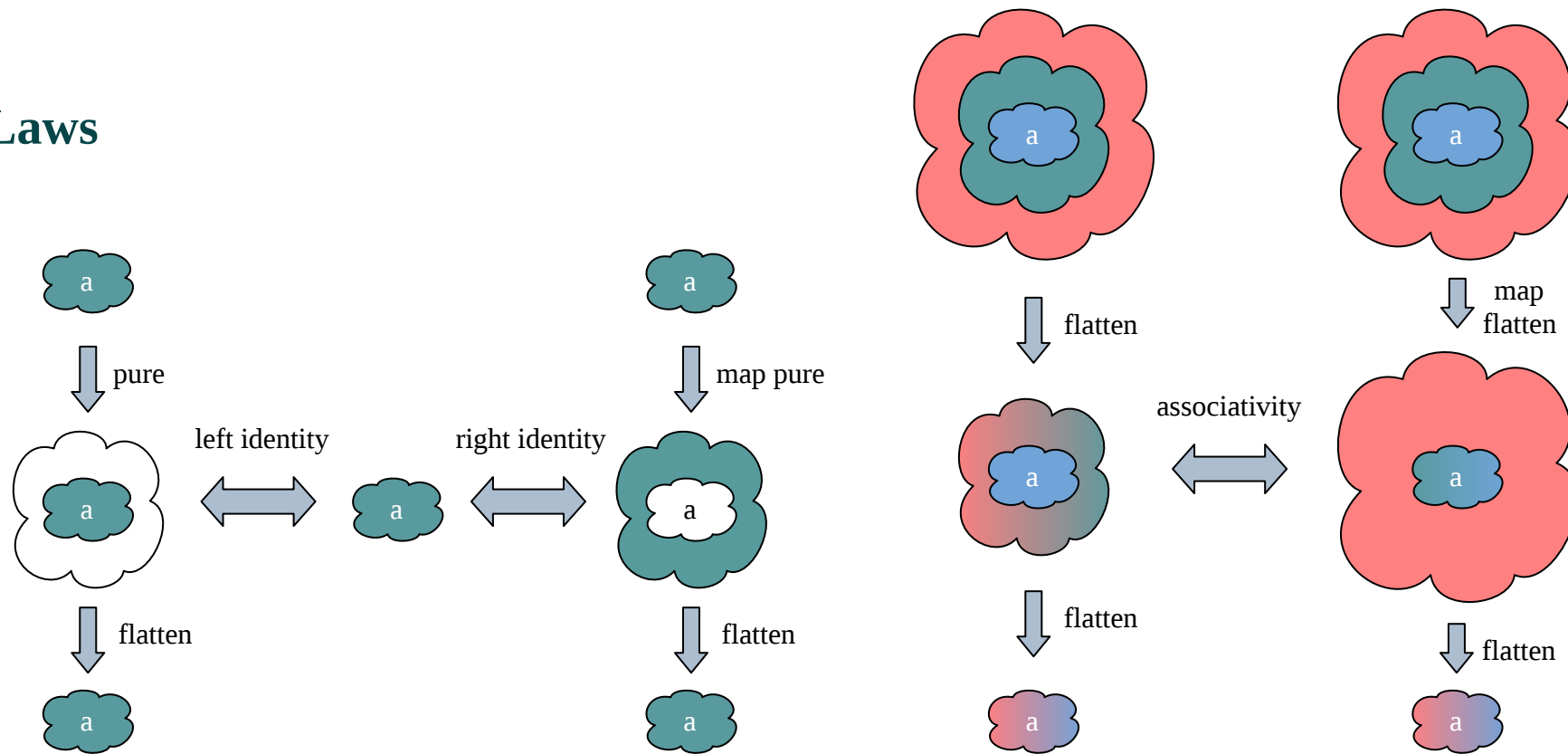
```
trait Monad[F[_]] extends Applicative[F] {  
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]  
  def flatten[A](ffa: F[F[A]]): F[A]  
  
  def flatTap[A, B](fa: F[A])(f: A => F[B]): F[A]  
  
  def forever[A](fa: F[A]): F[Nothing]  
}
```



## API



## Laws



# Exercise 3



# Monad is an Applicative

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

## Coherence Law

```
forAll((fa: F[A], fb: F[B]) =>  
  fa.flatMap(a => fb.map(b => (a, b))) == (fa, fb).tuple2  
)
```



# Monad is an Applicative

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

## Coherence Law

```
forAll{ (fa: F[A], fb: F[B]) =>  
  val combinedFor = for {  
    a <- fa  
    b <- fb  
  } yield (a, b)  
  combinedFor == (fa, fb).tuple2  
}
```



# Monad is for sequential composition of effects

```
val res =  
  for {  
    a <- fa  
    b <- foo(a)  
  } yield f(a,b)
```



# Monad is for sequential composition of effects

```
val res =  
  for {  
    a <- fa  
    b <- foo(a)  
  } yield f(a,b)
```

```
def sendStatement(userId: UserId): IO[Unit] =  
  for {  
    user    <- getUser(userId)  
    account <- getAccount(user.accountId)  
    _       <- sendStatement(user.email, account.statement)  
  } yield ()
```





Monad is for sequential composition

&

Applicative must be coherent with Monad



**All Monadic effects must have a sequential Applicative**



# Coherence implication

```
import answers.functors.FunctionsAnswers.DefaultMonad

def validateUsername(x: String): ValidatedNel[String, Username] = {
  if(x.length > 5) valid(Username(x))
  else invalidNel(s"Username $x too short")
}

def validateCountry(x: String): ValidatedNel[String, Country] =
  x match {
    case "FRA" => valid(Country.France)
    case _     => invalidNel(s"Unsupported country $x")
  }

implicit def validatedMonad[E]: Monad[Validated[E, ?]] = new DefaultMonad[Validated[E, ?]] {
  def pure[A](a: A): Validated[E, A] = Valid(a)

  def flatMap[A, B](fa: Validated[E, A])(f: A => Validated[E, B]): Validated[E, B] =
    fa match {
      case Invalid(e) => Invalid(e)
      case Valid(a)   => f(a)
    }
}
```



# Coherence implication

```
val combinedFor: ValidatedNel[String, User] = for {  
  username <- validateUsername("foo")  
  country  <- validateCountry("UK")  
} yield User(username, country)
```

```
scala> println(combinedFor)  
Invalid(NonEmptyList(Username foo too short))
```



# Coherence implication

```
val combinedFor: ValidatedNel[String, User] = for {  
  username <- validateUsername("foo")  
  country  <- validateCountry("UK")  
} yield User(username, country)
```

```
scala> println(combinedFor)  
Invalid(NonEmptyList(Username foo too short))
```

```
def map2[E: Semigroup, A, B, C](fa: Validated[E, A], fb: Validated[E, B])(f: (A, B) => C): Validated[E, C] =  
  (fa, fb) match {  
    case (Valid(a), Valid(b))      => Valid(f(a, b))  
    case (Invalid(e), Valid(_))    => Invalid(e)  
    case (Valid(_), Invalid(e))    => Invalid(e)  
    case (Invalid(e1), Invalid(e2)) => Invalid(e1 |+| e2)  
  }  
val combinedMap2 = map2(validateUsername("foo"), validateCountry("UK"))(User(_, _))
```

```
scala> println(combinedMap2)  
Invalid(NonEmptyList(Username foo too short, Unsupported country UK))
```



# Validated cannot be a Monad



**IO Applicative cannot combine in parallel (but IO.Par can)**



# Composition

if  $F$  and  $G$  are a Functor then  $F[G[_]]$  is a Functor

if  $F$  and  $G$  are an Applicative then  $F[G[_]]$  is an Applicative





# Composition

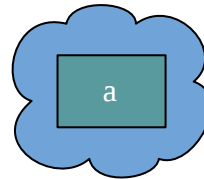
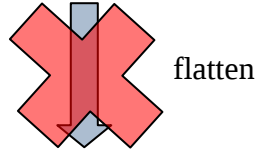
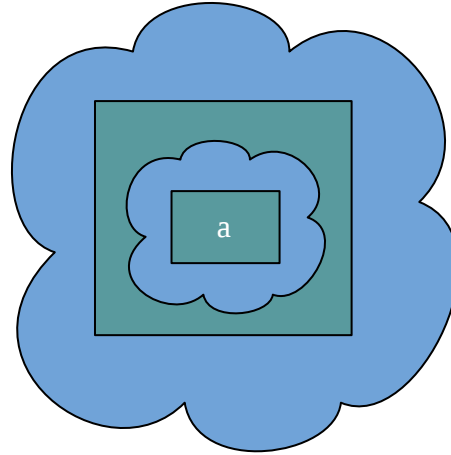
if  $F$  and  $G$  are a Functor then  $F[G[_]]$  is a Functor

if  $F$  and  $G$  are an Applicative then  $F[G[_]]$  is an Applicative

but if  $F$  and  $G$  are a Monad then  $F[G[_]]$  is not necessarily a Monad



# Monad do not compose



If you can combine effects then ...



# Imperative code is Monadic

```
def getOrderTotal(userId: Int, orderId: Int): Double = {  
  log.trace(s"User $userId get order total for order $orderId"): Unit  
  val hasAccess = canUserCanAccessOrder(userId, orderId): Boolean  
  if(! hasAccess)  
    throw new Exception(s"User $userId does not have access to order $orderId")  
  else {  
    val order = getOrder(orderId): Order  
    order.total: Double  
  }  
}
```



# Imperative code is Monadic

```
def getOrderTotal(userId: Int, orderId: Int): Double = {  
  log.trace(s"User $userId get order total for order $orderId"): Unit  
  val hasAccess = canUserCanAccessOrder(userId, orderId): Boolean  
  if(! hasAccess)  
    throw new Exception(s"User $userId does not have access to order $orderId")  
  else {  
    val order = getOrder(orderId): Order  
    order.total: Double  
  }  
}
```

```
type MyEffect[+A] = ... // Log + DB + Error with Exception  
  
def getOrderTotal(userId: Int, orderId: Int): MyEffect[Double] =  
  for {  
    _ <- log.trace(s"User $userId get order total for order $orderId"): MyEffect[Unit]  
    hasAccess <- canUserCanAccessOrder(userId, orderId): MyEffect[Boolean]  
    _ <- if(hasAccess) unit: MyEffect[Unit]  
        else fail(new Exception(s"User $userId does not have access to order $orderId")): MyEffect[Nothing]  
    order <- getOrder(orderId): MyEffect[Order]  
  } yield order.total: Double
```

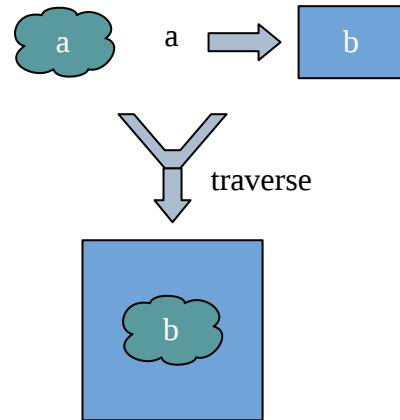


# Traverse



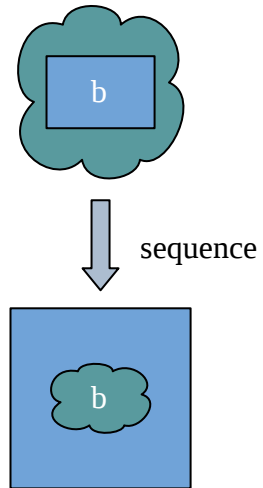
# Traverse

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {  
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]  
}
```



# Traverse alternative encoding

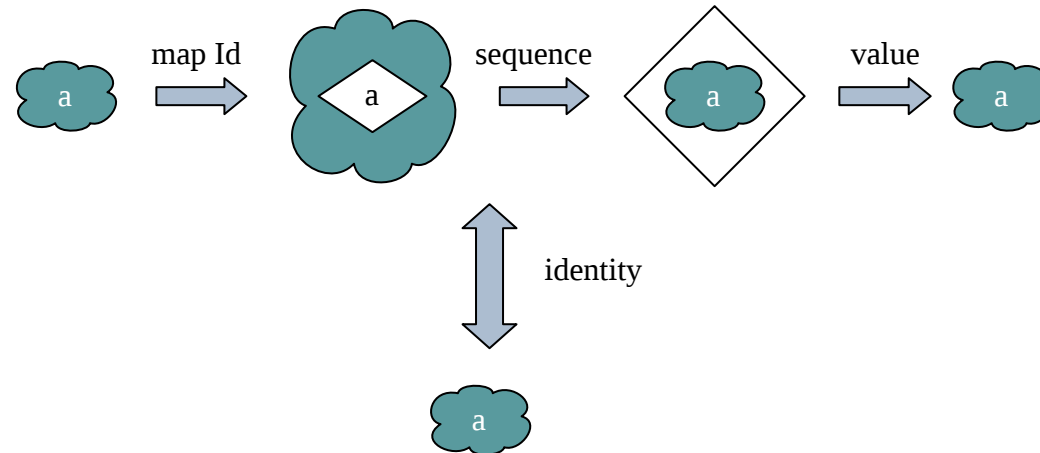
```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {  
  def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]]  
}
```





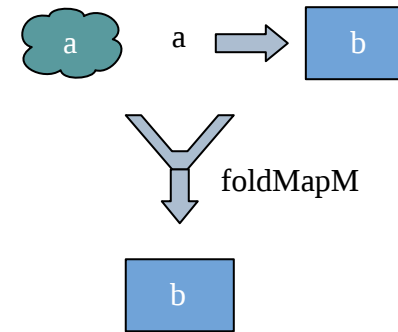
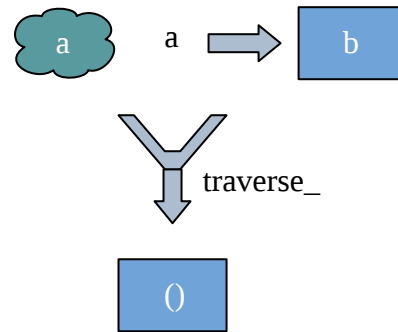
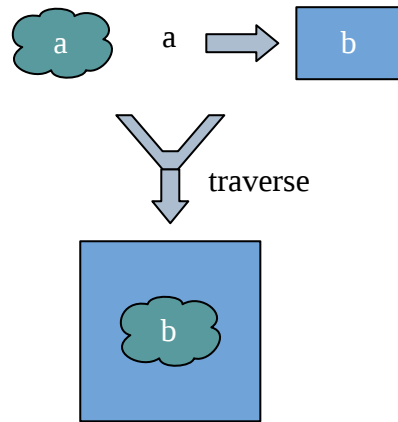
# Traverse Law

```
forAll(fa: F[A] => fa.map(Id(_)).sequence.value == fa)
```

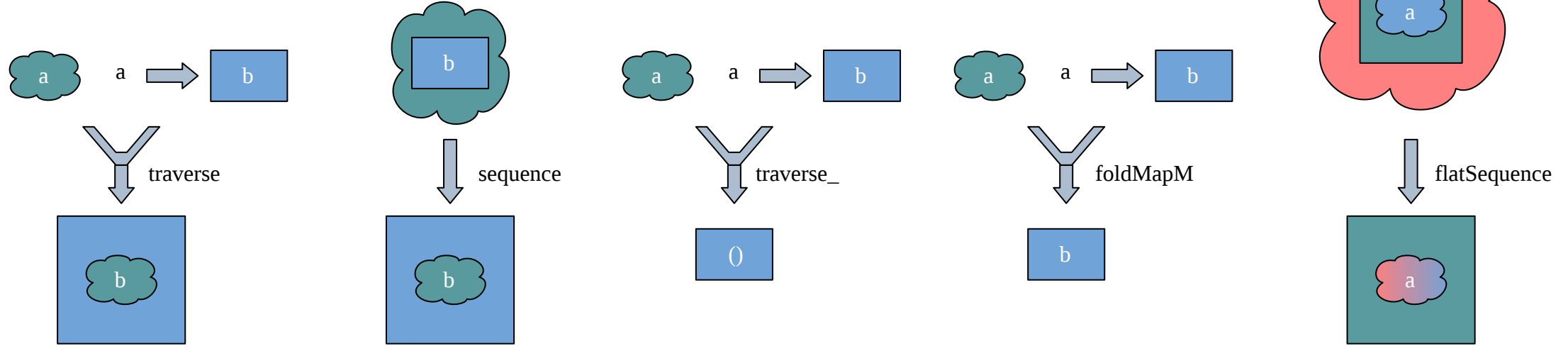


# Traverse API

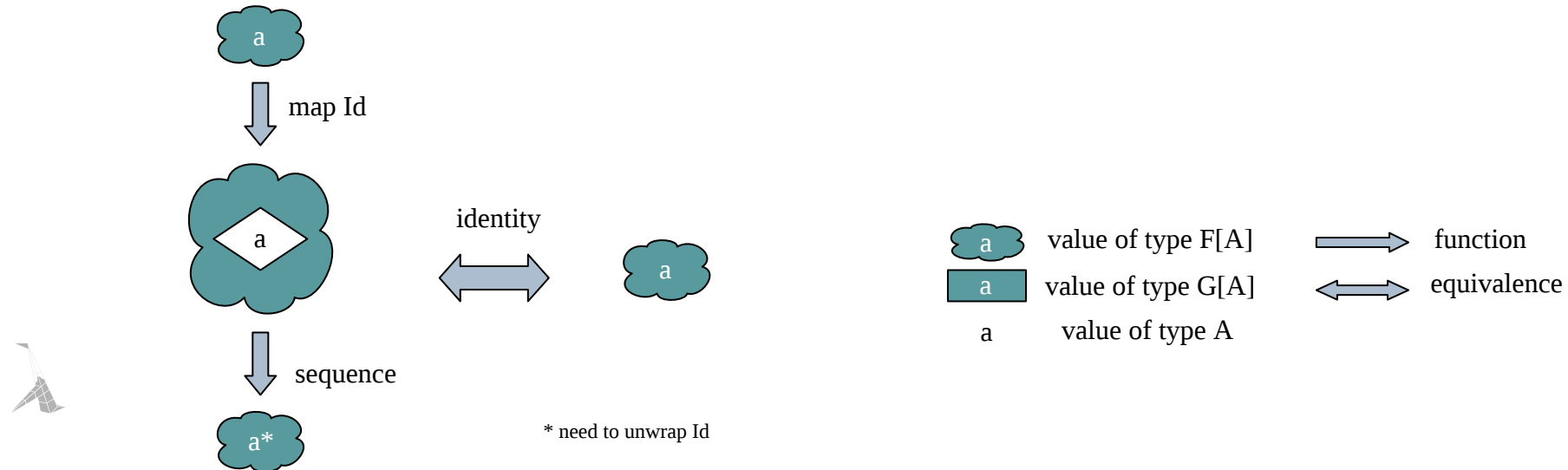
```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {  
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]  
  def traverse_[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[Unit]  
  def foldMapM[G[_]: Applicative, A, B: Monoid](fa: F[A])(f: A => G[B]): G[B]  
}
```



# API



# Laws



# Exercise 4



# Traverse is a Functor

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {  
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]  
}
```

## Coherence Law

```
forAll((fa: F[A], f: A => B) => fa.traverse(a => Id(f(a))).value == fa.map(f))
```



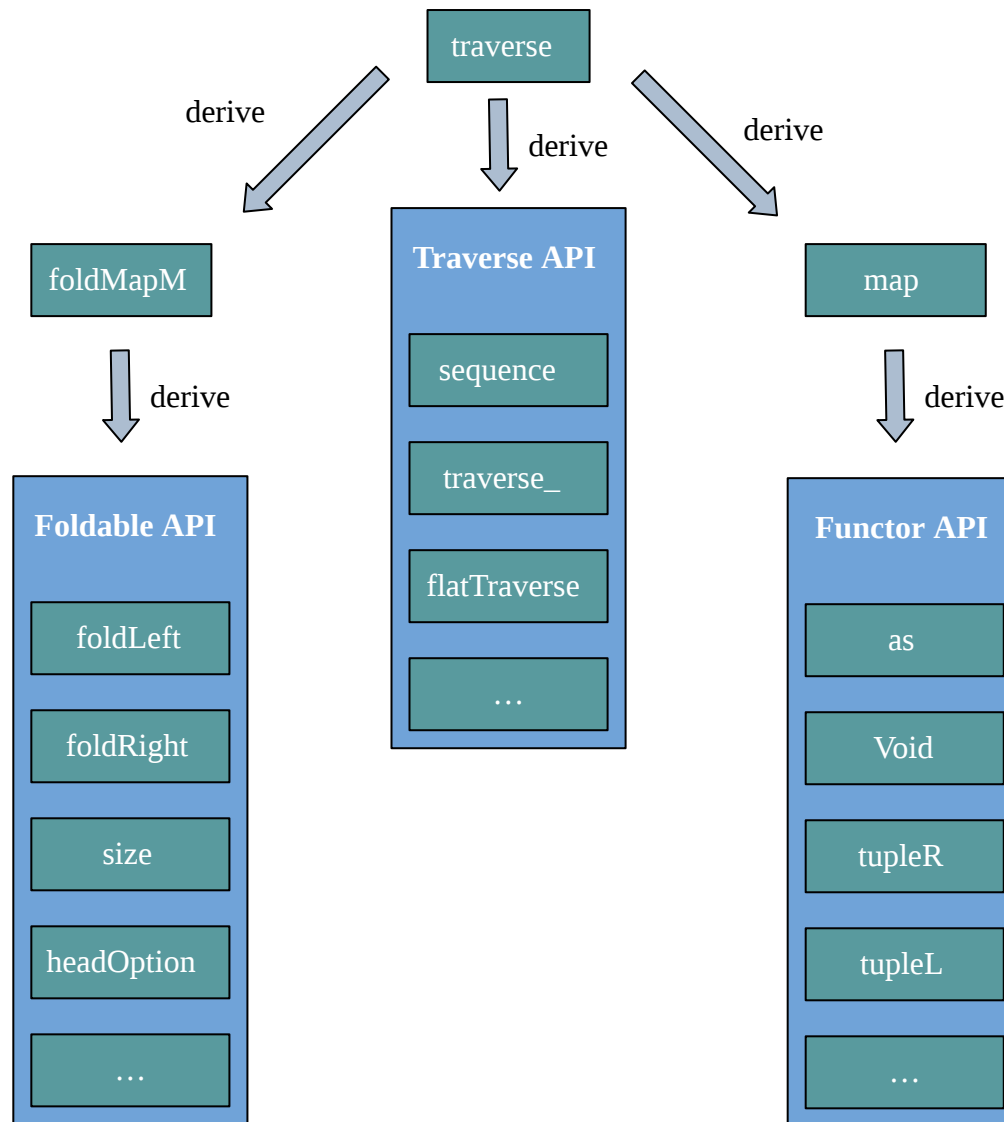
# Traverse is a Foldable

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {  
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]  
}
```

## Coherence Law

```
forAll((fa: F[A], f: A => B) => fa.traverse(a => Const(f(a))).getConst == fa.foldMap(f))
```





# Default implementation

```
traverse > foldMap > size
```

```
def sizeDefault[F[_]: Traverse, A](fa: F[A]): Int =  
  fa.traverse(_ => Const(1)).getConst
```

```
sizeDefault = (Const(1) |+| Const(1) |+| ... |+| Const(1)).getConst // O(n)
```





# Default implementation

```
traverse > foldMap > size
```

```
def sizeDefault[F[_]: Traverse, A](fa: F[A]): Int =  
  fa.traverse(_ => Const(1)).getConst
```

```
sizeDefault = (Const(1) |+| Const(1) |+| ... |+| Const(1)).getConst // 0(n)
```

```
val vectorTraverse: Traverse[Vector] = new Traverse[Vector] {  
  def traverse[G[_]: Applicative, A, B](fa: Vector[A])(f: A => G[B]): G[Vector[B]] = ???  
  overload def size[A](fa: Vector[A]): Int = fa.size // 0(1)  
}
```



# Default implementation

```
traverse > foldMap > size
```

```
def sizeDefault[F[_]: Traverse, A](fa: F[A]): Int =  
  fa.traverse(_ => Const(1)).getConst
```

```
sizeDefault = (Const(1) |+| Const(1) |+| ... |+| Const(1)).getConst // 0(n)
```

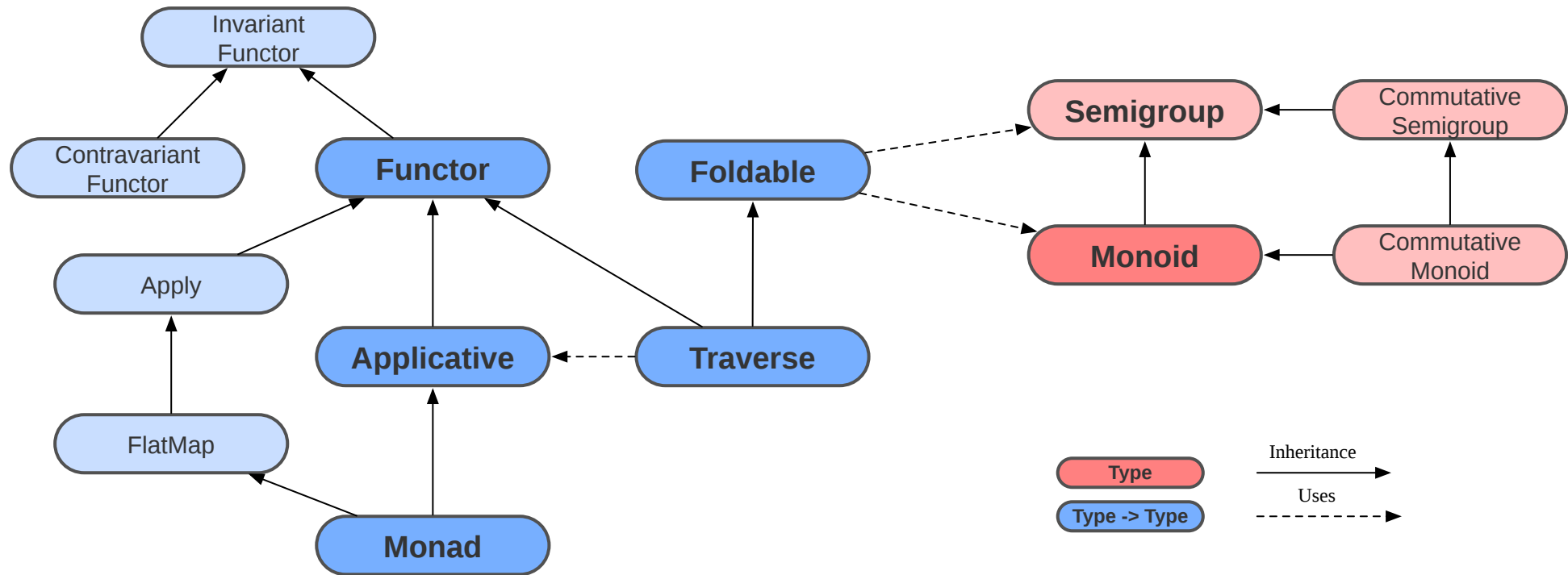
```
val vectorTraverse: Traverse[Vector] = new Traverse[Vector] {  
  def traverse[G[_]: Applicative, A, B](fa: Vector[A])(f: A => G[B]): G[Vector[B]] = ???  
  overload def size[A](fa: Vector[A]): Int = fa.size // 0(1)  
}
```

## Coherence law

```
forAll(fa: F[A] => fa.size == sizeDefault(fa))
```



# Review



# Granularity





# Granularity

- There is not a single "correct" granularity



# Granularity

- There is not a single "correct" granularity
- Applicative was discovered in 2008
- Selective Applicative Functor in 2018!



# Recommendations

## Define strongest typeclasses for data types

```
val instances: Monad[Option] with Traverse[Option] = ?
```

- Better defined behaviour
- More tests for free
- More powerful API

## Use weakest typeclasses for functions

```
def program[F[_]: Console: Applicative]: F[String] = ?
```

- Principle of least power
- More reasoning
- Less tests to write





# Resources and further study

- [Cats infographic](#): typeclass diagrams for cats
- [Thinking with types](#): variance
- [Constraints Liberate, Liberties Constrain](#) from Runar Bjarnason
- Applicative paper [Applicative Programming with Effects](#)
- Selective Applicative Functor paper [Selective Applicative Functors](#)



# The End

