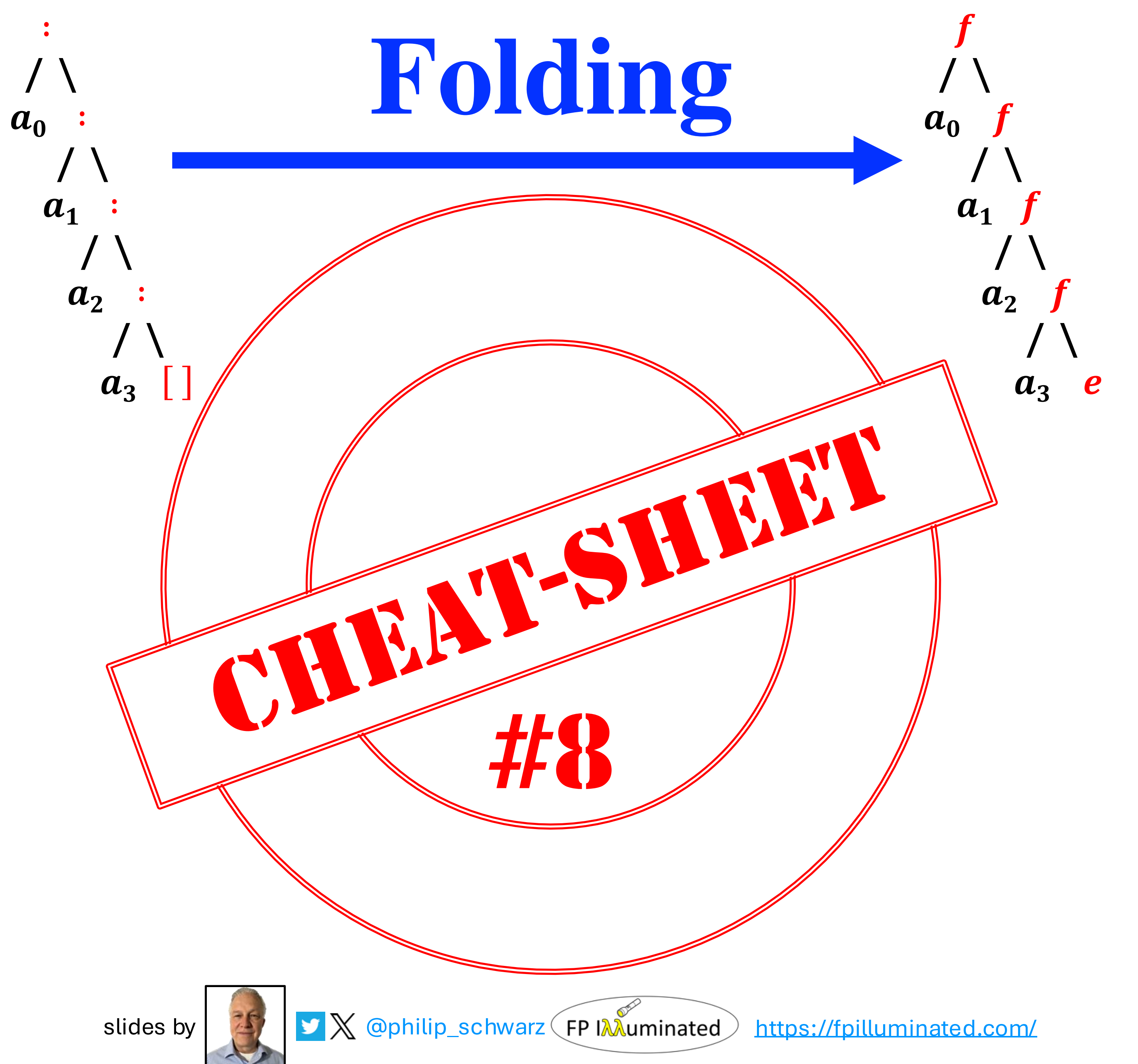


slides by @philip_schwarz FP Iλλuminated https://fpilluminated.com/

$foldl :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$
$foldl\ f\ b\ [\,] = b$
$foldl\ f\ b\ (x : xs) = foldl\ f\ (f\ b\ x)\ xs$

$foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$
$foldr\ f\ b\ [\,] = b$
$foldr\ f\ b\ (x : xs) = f\ x\ (foldr\ f\ b\ xs)$

First Duality Theorem of Fold

$foldr\ (\oplus)\ e\ xs = foldl\ (\oplus)\ e\ xs$

(for all finite lists xs)

when the pair $(\oplus, e)$ is a *Monoid*, i.e. for all x, y, and z we have

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$
$$e \oplus x = x \text{ and } x \oplus e = x$$

in other words, $\oplus$ is **associative** with **unit** e.

$e :: \alpha$
$(\oplus) :: \alpha \to \alpha \to \alpha$

$foldl' :: \alpha \to [\alpha] \to \alpha$
$foldl'\ a\ [\,] = a$
$foldl'\ a\ (x : xs) = foldl'\ (a \oplus x)\ xs$

$foldr' :: [\alpha] \to \alpha$
$foldr'\ [\,] = e$
$foldr'\ (x : xs) = x \oplus (foldr'\ xs)$

$fold :: Monoid\ \alpha \Rightarrow [\alpha] \to \alpha$
$fold\ [\,] = mempty$
$fold\ (x : xs) = x\ `mappend`\ fold\ xs$

$mempty :: \alpha$
$mappend :: \alpha \to \alpha \to \alpha$

$1_A : A$
$(\oplus) :: A \times A \to A$

Monoid

A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$
1_A: identity for $\oplus$

Identity: $x = x \oplus 1_A = 1_A \oplus x$
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

$$
\begin{aligned}
&fold && :: Monoid\ \alpha \Rightarrow [\alpha] \rightarrow \alpha \\
&fold\ [\,] && = mempty \\
&fold\ (x : xs) && = x\ 'mappend'\ fold\ xs
\end{aligned}
$$

$$
\begin{aligned}
&mempty :: \alpha \\
&mappend :: \alpha \ \rightarrow \alpha \ \rightarrow \alpha
\end{aligned}
$$

$$
\begin{aligned}
&foldMap && :: Monoid\ \beta \Rightarrow (\alpha \ \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta \\
&foldMap\ _\ [\,] && = mempty \\
&foldMap\ f\ (x : xs) && = f\ x\ 'mappend'\ foldMap\ f\ xs
\end{aligned}
$$

$foldMap$ generalises $fold$ by taking function $f :: (\alpha \ \rightarrow \beta)$ as an additional argument, which is applied to each list element prior to combining the resulting values using the $mempty$ and $mappend$ functions of $Monoid\ \beta$.

$$fold :: Monoid\ \alpha \Rightarrow [\alpha] \rightarrow \alpha$$
$$fold\ [\,] = mempty$$
$$fold\ (x : xs) = x\ `mappend`\ fold\ xs$$

$$mempty :: \alpha$$
$$mappend :: \alpha \rightarrow \alpha \rightarrow \alpha$$

$$foldMap :: Monoid\ \beta \Rightarrow (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$$
$$foldMap\ _\ [\,] = mempty$$
$$foldMap\ f\ (x : xs) = f\ x\ `mappend`\ foldMap\ f\ xs$$

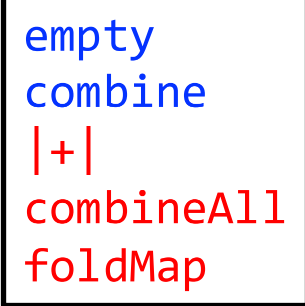
mappend
is a synonym for
(<>)

$$fold :: Monoid\ \alpha \Rightarrow [\alpha] \rightarrow \alpha$$
$$fold\ [\,] = mempty$$
$$fold\ (x : xs) = x <> fold\ xs$$

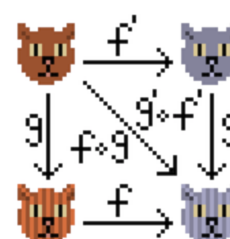
$$mempty :: \alpha$$
$$(<>) :: \alpha \rightarrow \alpha \rightarrow \alpha$$

$$foldMap :: Monoid\ \beta \Rightarrow (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$$
$$foldMap\ _\ [\,] = mempty$$
$$foldMap\ f\ (x : xs) = f\ x <> foldMap\ f\ xs$$

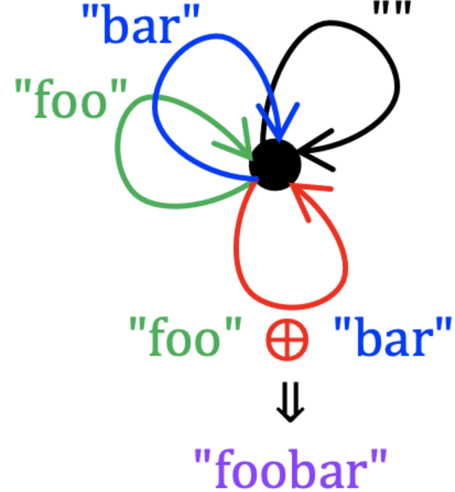
```
mempty
mappend
<>
fold
foldMap
```



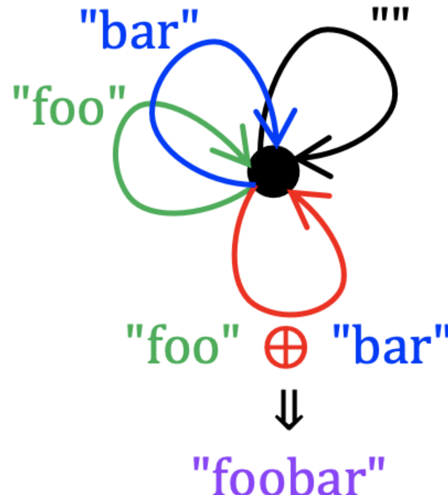
Cats



$\{A = \text{String} = [Char], \oplus = +\!\!+, 1_A = ""\}$



$\{A = \text{String}, \oplus = +\!\!+, 1_A = ""\}$



```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```haskell
> mempty::String
""

> "foo" <> mempty
"foo"

> mempty <> "bar"
"bar"

> "foo" <> "bar"
"foobar"

> mempty <> mempty::String
""

> fold ([]::[String])
""

> fold ["foo", "bar", "baz"]
"foobarbaz”

> foldMap (\s -> fmap toUpper s) ["foo", "bar", "baz"]
"FOOBARBAZ"
```

```scala
> val empty = Monoid[String].empty
val empty: String = ""

> "foo" |+| empty
val res1: String = foo

> empty |+| "bar"
val res2: String = bar

> "foo" |+| "bar"
val res3: String = foobar

> empty |+| empty
val res4: String = ""

> List.empty[String].combineAll
val res5: String = ""

> List("foo", "bar", "baz").combineAll
val res6: String = foobarbaz

> List("foo", "bar", "baz").foldMap(_.toUpperCase)
val res7: String = FOOBARBAZ
```

$\{A = Sum\ Int, \oplus = +, 1_A = 0\}$

$Sum\ 3$

$Sum\ 2$

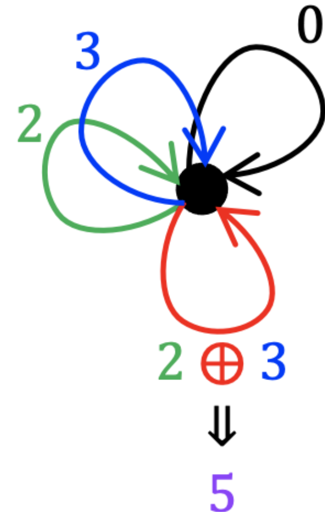
$Sum\ 0$

$Sum\ 2 \oplus Sum\ 3$

$\Downarrow$

$Sum\ 5$

$\{A = Int, \oplus = +, 1_A = 0\}$



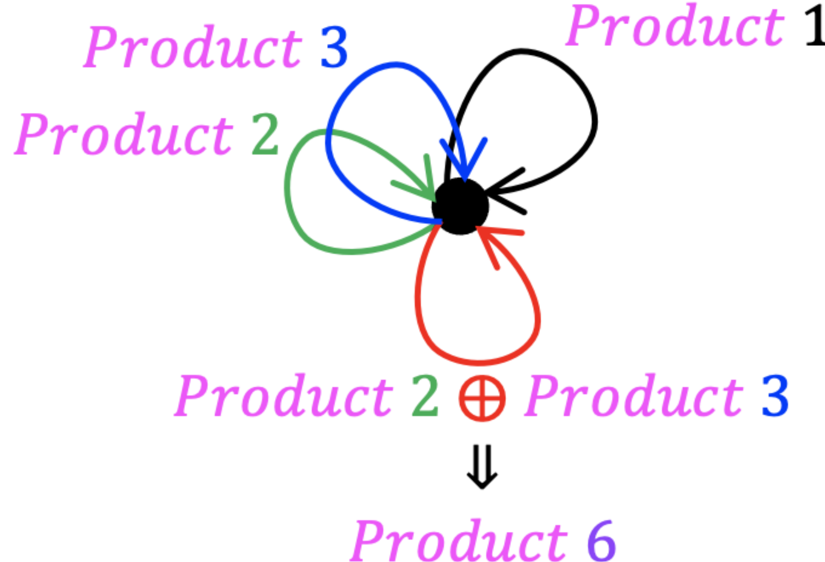
```haskell
newtype Sum a = Sum {getSum :: a}
```

```haskell
> getSum (mempty::Sum Int)
0

> getSum (Sum 2 <> mempty)
2

> getSum (mempty <> Sum 3)
3

> getSum (Sum 2 <> Sum 3)
5

> getSum (mempty <> mempty::Sum Int)
0

> getSum (fold ([]::[Sum Int]))
0

> getSum (fold (fmap Sum [1, 2, 3, 4]))
10

> getSum (foldMap Sum [1, 2, 3, 4])
10
```

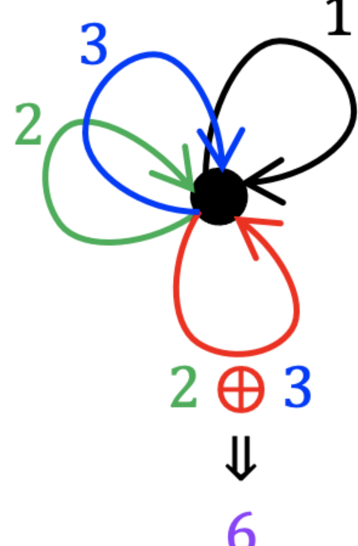
```scala
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```scala
> val empty = Monoid[Int].empty
val empty: Int = 0

> 2 |+| empty
val res1: Int = 2

> empty |+| 3
val res2: Int = 3

> 2 |+| 3
val res3: Int = 5

> empty |+| empty
val res4: Int = 0

> List.empty[Int].combineAll
val res5: Int = 0

> List(1, 2, 3, 4).combineAll
val res6: Int = 10

> List(1, 2, 3, 4).foldMap(_ + 10)
val res7: Int = 50
```

$\{A = \mathit{Product}\ \mathit{Int}, \oplus = \times, 1_A = 1\}$



$\{A = \mathit{Int}, \oplus = \times, 1_A = 1\}$



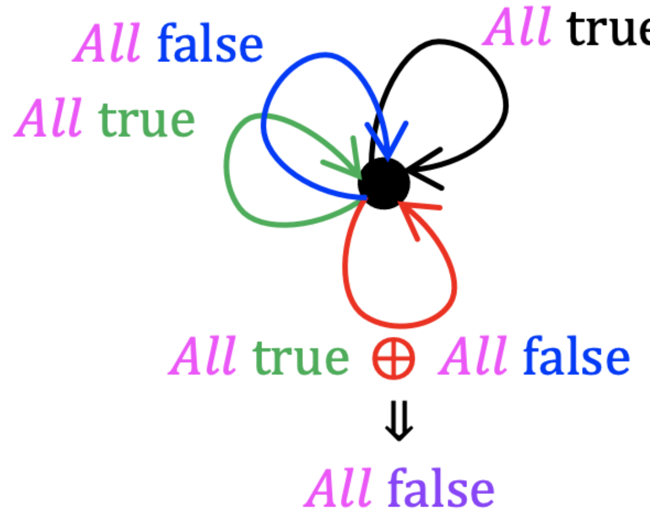
```haskell
newtype Product a = Product {getProduct :: a}
```

```haskell
> getProduct (mempty::Product Int)
1

> getProduct (Product 2 <> mempty)
2

> getProduct (mempty <> Product 3)
3

> getProduct (Product 2 <> Product 3)
6

> getProduct (mempty <> mempty::Product Int)
1

> getProduct (fold ([]::[Product Int]))
1

> getProduct (fold (fmap Product [1, 2, 3, 4]))
24

> getProduct (foldMap Product [1, 2, 3, 4])
24
```

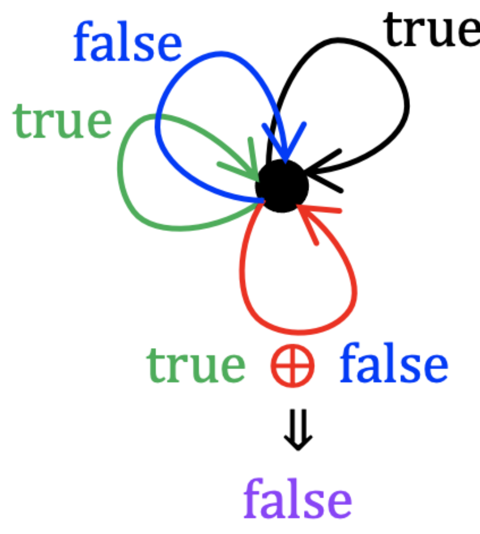
```scala
> import cats.implicits.catsSyntaxSemigroup
> import cats.syntax.foldable.*
> import cats.Monoid
```

```scala
> given Monoid[Int] =
   Monoid.instance(emptyValue = 1, cmb = _ * _)

> val empty = Monoid[Int].empty
val empty: Int = 1

> 2 |+| empty
val res1: Int = 2

> empty |+| 3
val res2: Int = 3

> 2 |+| 3
val res3: Int = 6

> empty |+| empty
val res4: Int = 1

> List.empty[Int].combineAll
val res5: Int = 1

> List(1, 2, 3, 4).combineAll
val res6: Int = 24

> List(1, 2, 3, 4).foldMap(_ + 10)
val res7: Int = 24024
```

$\{A = \mathit{All}\ \mathit{Bool}, \oplus = \&\&, 1_A = \text{true}\}$



$\{A = \text{Boolean}, \oplus = \&\&, 1_A = \text{true}\}$



```
newtype All = All {getAll :: Bool}
```

```
> getAll (mempty::All)
True

> getAll (All True <> mempty)
True

> getAll (All False <> mempty)
False

> getAll (mempty <> All True)
True

> getAll (mempty <> All False)
False

> getAll (mempty <> mempty::All)
True

> getAll (All False <> All False)
False
```

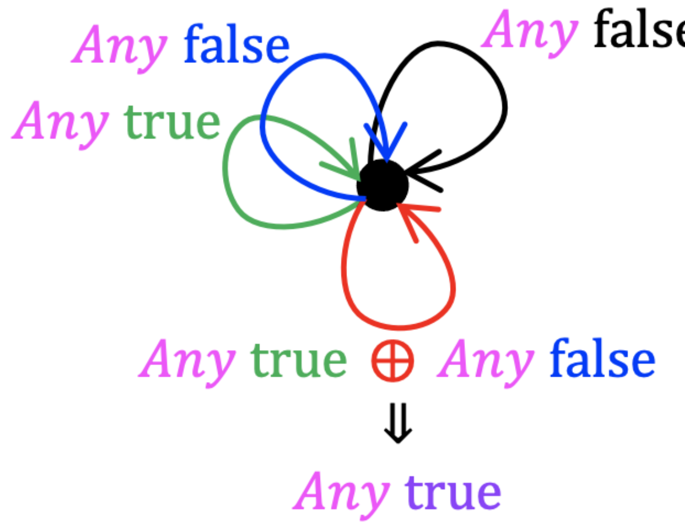
```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```
> given Monoid[Boolean] =
   Monoid.instance(emptyValue = true, cmb = _ && _)

> val empty = Monoid[Boolean].empty
val empty: Boolean = true

> true |+| empty
val res1: Boolean = true

> false |+| empty
val res2: Boolean = false

> empty |+| true
val res3: Boolean = true

> empty |+| false
val res4: Boolean = false

> empty |+| empty
val res5: Boolean = true

> false |+| false
val res6: Boolean = false
```

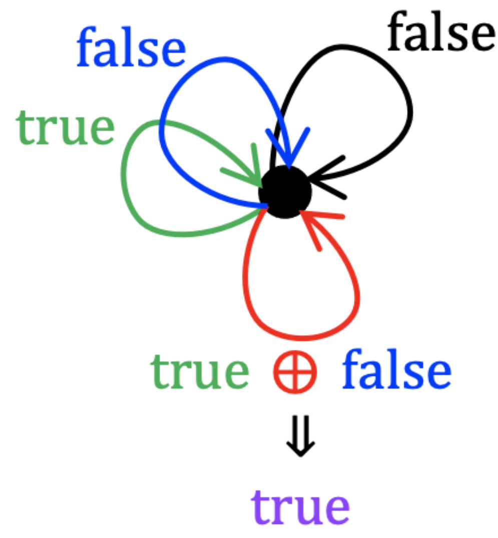
```
> getAll (fold ([]::[All]))
True

> getAll (fold (fmap All [True, True, True]))
True

> getAll (fold (fmap All [True, False, True]))
False

> getAll (foldMap All [True, True, True])
True

> getAll (foldMap All [True, True, False])
False
```

```
> List.empty[Boolean].combineAll
val res7: Boolean = true

> List(true, true, true).combineAll
val res8: Boolean = true

> List(true, false, true).combineAll
val res9: Boolean = false

> List(false, false, false).foldMap(!_)
val res10: Boolean = true

> List(false, true, false).foldMap(!_)
val res11: Boolean = false
```

$\{A = \mathit{Any\ Bool}, \oplus = ||, 1_A = \text{false}\}$



$\{A = \text{Boolean}, \oplus = ||, 1_A = \text{false}\}$



```
newtype Any = Any {getAny :: Bool}
```

```
> getAny (mempty::Any)
False

> getAny (Any True <> mempty)
True

> getAny (Any False <> mempty)
False

> getAny (mempty <> Any True)
True

> getAny (mempty <> Any False)
False

> getAny (mempty <> mempty:: Any)
False

> getAny (Any True <> Any True)
True
```

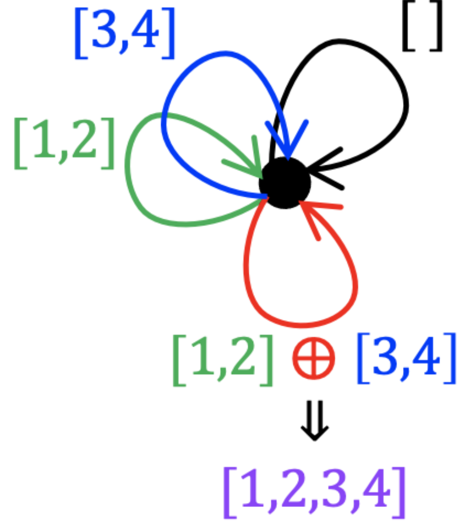
```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```
> given Monoid[Boolean] =
   Monoid.instance(emptyValue = false, cmb = _ || _)

> val empty = Monoid[Boolean].empty
val empty: Boolean = false

> true |+| empty
val res1: Boolean = true

> false |+| empty
val res2: Boolean = false

> empty |+| true
val res3: Boolean = true

> empty |+| false
val res4: Boolean = false

> empty |+| empty
val res5: Boolean = false

> true |+| true
val res6: Boolean = true
```

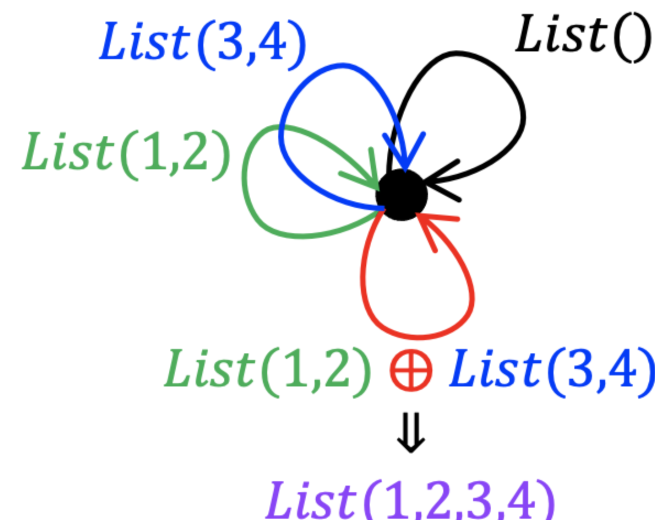
```haskell
> getAny (fold ([]::[Any]))
False

> getAny (fold (fmap Any [False, True, False]))
True

> getAny (fold (fmap Any [False, False, False]))
False

> getAny (foldMap Any [False, True, False])
True

> getAny (foldMap Any [False, False, False])
False
```

```scala
> List.empty[Boolean].combineAll
val res7: Boolean = false

> List(false, true, false).combineAll
val res8: Boolean = true

> List(false, false, false).combineAll
val res9: Boolean = false

> List(true, false, true).foldMap(!_)
val res10: Boolean = true

> List(true, true, true).foldMap(!_)
val res11: Boolean = false
```

$\{A = [\mathrm{Int}], \oplus = +\!\!+, 1_A = \mathrm{Nil}\}$



$\{A = \mathrm{List(Int)}, \oplus = +\!\!+, 1_A = \mathrm{Nil}\}$



```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```haskell
> mempty::[Int]
[]

> [1,2] <> mempty
[1,2]

> mempty <> [3,4]
[3,4]

> [1,2] <> [3,4]
[1,2,3,4]

> mempty <> mempty::[Int]
[]

> fold (mempty::[[Int]])
[]

> fold [[1,2],[3,4,5],[6]]
[1,2,3,4,5,6]

> foldMap tail [[1,2],[3,4,5],[6]]
[2,4,5]
```

```scala
> val empty = Monoid[List[Int]].empty
val empty: List[Int] = List()

> List(1,2) |+| empty
val res1: List[Int] = List(1, 2)

> empty |+| List(3,4)
val res2: List[Int] = List(3, 4)

> List(1,2) |+| List(3,4)
val res3: List[Int] = List(1, 2, 3, 4)

> empty |+| empty
val res4: List[Int] = List()

> List.empty[List[Int]].combineAll
val res5: List[Int] = List()

> List(List(1,2),List(3,4,5),List(6)).combineAll
val res6: List[Int] = List (1, 2, 3, 4, 5, 6)

> List(List(1,2),List(3,4,5),List(6)).foldMap(_.tail)
val res7: List[Int] = List(2, 4, 5)
```

```haskell
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> b       = b
    a       <> Nothing = a
    Just a  <> Just b  = Just (a <> b)

instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

```scala
class OptionMonoid[A]
  (implicit A: Semigroup[A]) extends Monoid[Option[A]] {
  def empty: Option[A] = None
  def combine(x: Option[A], y: Option[A]): Option[A] =
    x match {
      case None => y
      case Some(a) =>
        y match {
          case None    => x
          case Some(b) => Some(A.combine(a, b)) } } }
```

$\{A = \text{Maybe } (\text{Sum Int}), \oplus = \text{mappend} = \text{<>}, 1_A = \text{Nothing}\}$

$\{A = \textit{Sum Int}, \oplus = +, 1_A = 0\}$

$\{A = \text{Option[Int]}, \oplus = \text{combine} = |+|, 1_A = \text{None}\}$

$\{A = \textit{Int}, \oplus = +, 1_A = 0\}$

Just (*Sum* 3)

Just (*Sum* 2)

Nothing

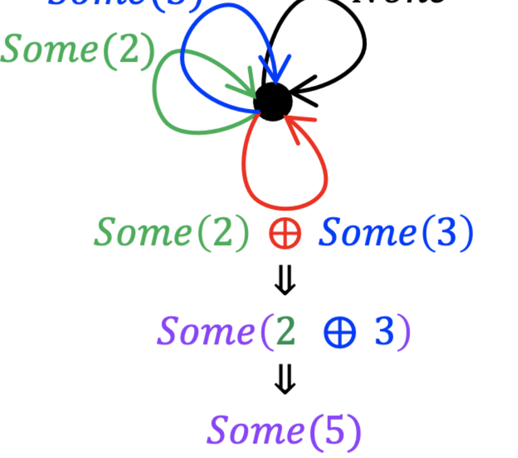
Just (*Sum* 2) ⊕ *Just* (*Sum* 3)

⇓

Just ((*Sum* 2) ⊕ (*Sum* 3))

⇓

Just (*Sum* 5)



```
> mempty :: Maybe (Sum Int)
Nothing

> Just (Sum 2) <> Just (Sum 3)
Just (Sum {getSum = 5})

> Just (Sum 2) <> mempty
Just (Sum {getSum = 2})

> mempty <> Just (Sum 3)
Just (Sum {getSum = 3})

>(mempty :: Maybe (Sum Int)) <> mempty
Nothing

> fold ([]::[Maybe (Sum Int)])
Nothing

> fold [Just (Sum 1), Nothing, Just (Sum 2), Nothing, Just (Sum 3), Just (Sum 4)]
Just (Sum {getSum = 10})

> fold (fmap (\x -> fmap Sum x) [Just 1, Nothing, Just 2, Nothing, Just 3, Just 4])
Just (Sum {getSum = 10})

> foldMap (\x -> fmap Sum x) [Just 1, Nothing, Just 2, Nothing, Just 3, Just 4]
Just (Sum {getSum = 10})
```

```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```
> val empty = Monoid[Option[Int]].empty
val empty: Option[Int] = None

> 2.some |+| 3.some
val res1: Option[Int] = Some(5)

> 2.some |+| empty
val res2: Option[Int] = Some(2)

> empty |+| 3.some
val res3: Option[Int] = Some(3)

> empty |+| empty
val res4: Option[Int] = None

> List.empty[Option[Int]].combineAll
val res5: Option[Int] = None

> List(1.some, none, 2.some, none, 3.some, 4.some).combineAll
val res6: Option[Int] = Some(10)

> List(1.some, none, 2.some, none, 3.some, 4.some).foldMap(_.map(_ * 10))
val res7: Option[Int] = Some(100)
```

$\{A = [\mathrm{Int}], \oplus = +\!\!+, 1_A = \mathrm{Nil}\}$

$\{A = \mathrm{Maybe}\ [\mathrm{Int}], \oplus = \mathrm{mappend} = <>, 1_A = \mathrm{Nothing}\}$

Just [3, 4]

Nothing

Just [1, 2]

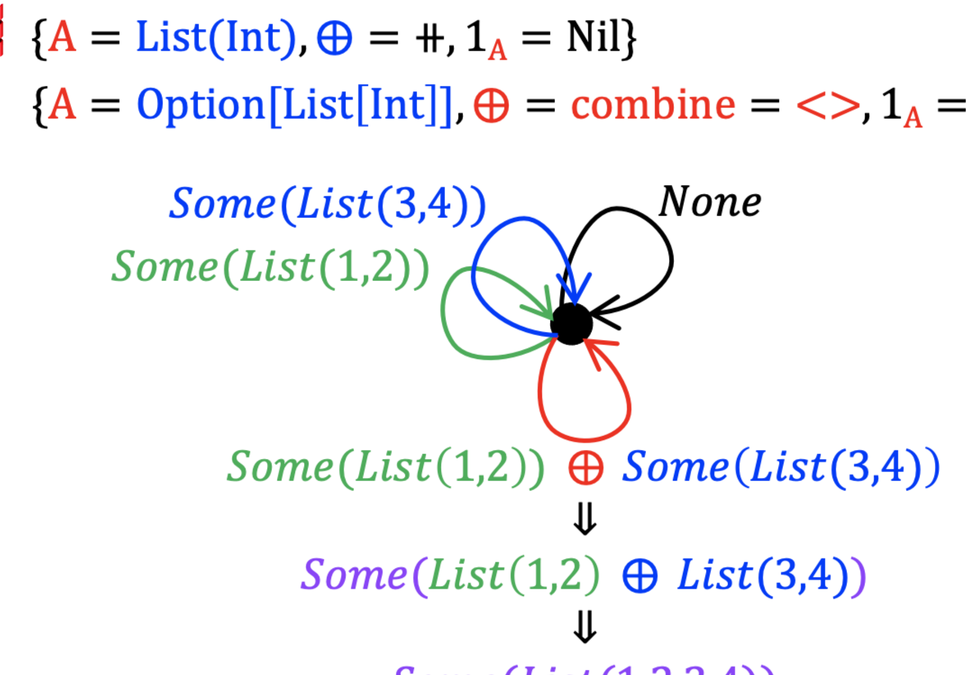
Just [1, 2] ⊕ *Just* [3, 4]]

⇓

Just ([1,2] ⊕ [3,4])

⇓

Just [1,2,3,4]



```
> mempty :: Maybe [Int]
Nothing

> Just [1,2] <> Just [3,4]
Just [1,2,3,4]

> Just [1,2] <> mempty
Just [1,2]

> mempty <> Just [3,4]
Just [3,4]

>(mempty :: Maybe [Int]) <> mempty
Nothing

> fold ([]::[Maybe [Int]])
Nothing

> fold [Just [1,2], Nothing, Just [3,4,5], Nothing, Just [6]]
Just [1,2,3,4,5,6]

> foldMap (\x -> fmap tail x) [Just [1,2], Nothing, Just [3,4,5], Nothing, Just [6]]
Just [2,4,5]
```

```
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```
> val empty = Monoid[Option[List[Int]]].empty
val empty: Option[List[Int]] = None

> List(1,2).some |+| List(3,4).some
val res1: Option[List[Int]] = Some(List(1,2,3,4))

> List(1,2).some |+| empty
val res2: Option[List[Int]] = Some(List(1,2))

> empty |+| List(3,4).some
val res3: Option[List[Int]] = Some(List(3,4))

> empty |+| empty
val res4: Option[List[Int]] = None

> List.empty[Option[List[Int]]].combineAll
val res5: Option[List[Int]] = None

> List(List(1,2).some , none, List(3,4,5).some, none, List(6).some).combineAll
val res6: Option[List[Int]] = Some(List(1,2,3,4,5,6))

> List(List(1,2).some , none, List(3,4,5).some, none, List(6).some).foldMap(_.map(_.tail))
val res7: Option[List[Int]] = Some(List(2,4,5))
```

$\{A = \text{Map Currency } (\mathit{Sum} \text{ Money}), \oplus = \text{monoidal accumulation}, 1_A = \text{Map.mempty}\}$

$\{A = \mathit{Sum\ Int}, \oplus = +, 1_A = 0\}$

$\mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ 2), (\mathit{GBP}, \mathit{Sum}\ 1)]$

$\mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ 3), (\mathit{EUR}, \mathit{Sum}\ 4)]$

Map. mempty

$\mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ 2), (\mathit{GBP}, \mathit{Sum}\ 1)] \oplus \mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ 3), (\mathit{EUR}, \mathit{Sum}\ 4)]$

$\Downarrow$

$\mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ (2) \oplus \mathit{Sum}\ (3)), (\mathit{GBP}, \mathit{Sum}\ 1), (\mathit{EUR}, \mathit{Sum}\ 4)]$

$\Downarrow$

$\mathit{Map.fromList}\ [(\mathit{USD}, \mathit{Sum}\ 5), (\mathit{GBP}, \mathit{Sum}\ 1), (\mathit{EUR}, \mathit{Sum}\ 4)]$

Data.HashMap.Monoidal

```
newtype MonoidalHashMap k a
```

This module provides a **HashMap** variant which uses the value's **Monoid** instance to **accumulate** conflicting entries when merging Maps.

```
…
import qualified Data.HashMap.Strict as M
import Data.Hashable (Hashable)
…
-- | A 'HashMap' with monoidal accumulation
newtype MonoidalHashMap k a = MonoidalHashMap { getMonoidalHashMap :: M.HashMap k a }
    deriving ( Show, Read, Functor, Eq, …)
…
instance (Eq k, Hashable k, Semigroup a) => Semigroup (MonoidalHashMap k a) where
    MonoidalHashMap a <> MonoidalHashMap b = MonoidalHashMap $ M.unionWith (<>) a b
…

instance (Eq k, Hashable k, Semigroup a) => Monoid (MonoidalHashMap k a) where
mempty = MonoidalHashMap mempty
…
mappend (MonoidalHashMap a) (MonoidalHashMap b) = MonoidalHashMap $ M.unionWith (<>) a b
…
```

https://hackage.haskell.org/package/monoidal-containers-0.6.5.0/docs/Data-HashMap-Monoidal.html

```
> {-# LANGUAGE DeriveGeneric #-}
> import Data.HashMap.Monoidal (MonoidalHashMap)
> import qualified Data.HashMap.Monoidal as MonoidalHashMap

> data Currency = EUR | USD | GBP deriving (Eq, Ord, Enum, Show, Generic)
> type Money = Int
> instance Hashable Currency
> type Account  = MonoidalHashMap Currency Money

> account1 = MonoidalHashMap.fromList [(USD, 10), (GBP, 5), (EUR, 1)]
> account2 = MonoidalHashMap.fromList [(GBP, 2)]
> account3 = MonoidalHashMap.fromList [(USD, 3), (EUR, 5)]

> mempty :: MonoidalHashMap Currency (Sum Money)
MonoidalHashMap {getMonoidalHashMap = fromList []}

> fmap getSum ((fmap Sum account1) <> mempty)
MonoidalHashMap {getMonoidalHashMap = fromList [(EUR,1),(GBP,5),(USD,10)]}

> fmap getSum ((fmap Sum account1) <> (fmap Sum account2))
MonoidalHashMap {getMonoidalHashMap = fromList [(EUR,1),(GBP,7),(USD,10)]}

> (fmap getSum (fold (fmap (fmap Sum) [account1, account2, account3])))
MonoidalHashMap {getMonoidalHashMap = fromList [(EUR,6),(GBP,7),(USD,13)]}

> (fmap getSum (foldMap (fmap Sum) [account1, account2, account3]))
MonoidalHashMap {getMonoidalHashMap = fromList [(EUR,6),(GBP,7),(USD,13)]}
```

$\{A = \text{Map[Currency, Money]}, \oplus = \text{monoidal accumulation}, 1_A = \text{Map.empty}\}$

$\{A = Int, \oplus = +, 1_A = 0\}$

$Map(USD \to 2, GBP \to 1)$

$Map(USD \to 3, EUR \to 4)$

Map. empty

$$Map(USD \to 2, GBP \to 1) \oplus Map(USD \to 3, EUR \to 4)$$

$$\Downarrow$$

$$Map(USD \to 2 \oplus 3, GBP \to 1, EUR \to 4)$$

$$\Downarrow$$

$$Map(USD \to 5, GBP \to 1, EUR \to 4)$$

```scala
class MapMonoid[K, V](implicit V: Semigroup[V]) extends Monoid[Map[K, V]] {
  def empty: Map[K, V] = Map.empty

  def combine(xs: Map[K, V], ys: Map[K, V]): Map[K, V] =
    if (xs.size <= ys.size) {
      xs.foldLeft(ys) { case (my, (k, x)) =>
        my.updated(k, Semigroup.maybeCombine(x, my.get(k)))
      }
    } else {
      ys.foldLeft(xs) { case (mx, (k, y)) =>
        mx.updated(k, Semigroup.maybeCombine(mx.get(k), y))
      }
    }

  override def combineAll(xss: IterableOnce[Map[K, V]]): Map[K, V] = …
  …
```

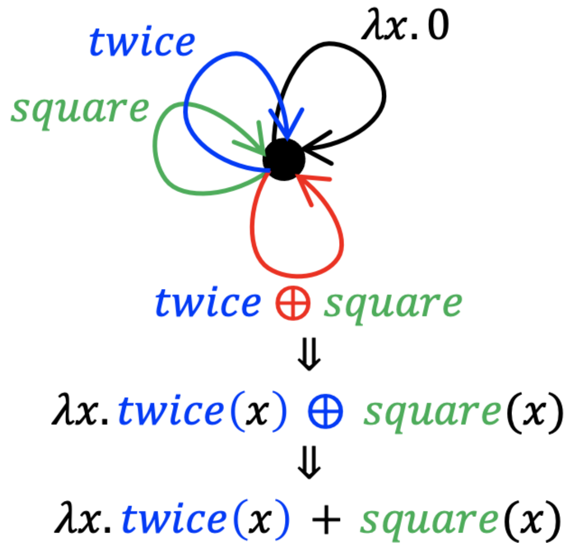
Cats

```scala
abstract class SemigroupFunctions[S[T] <: Semigroup[T]] {
  …
  def maybeCombine[@sp(Int, Long, Float, Double) A](ox: Option[A], y: A)(implicit ev: S[A]): A = ox match {
    case Some(x) => ev.combine(x, y)
    case None    => y
  }

  def maybeCombine[@sp(Int, Long, Float, Double) A](x: A, oy: Option[A])(implicit ev: S[A]): A = oy match {
    case Some(y) => ev.combine(x, y)
    case None    => x
  }
  …
```

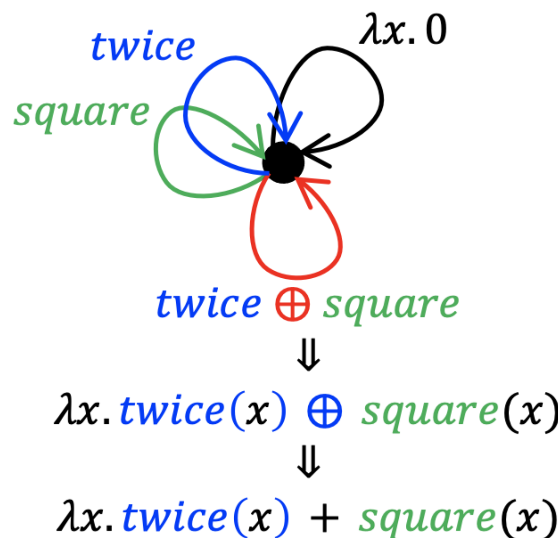
https://typelevel.org/cats/api/cats/kernel/instances/MapMonoid.html

```
> enum Currency { case EUR, USD, GBP }
> type Money = Int
> type Account = Map[Currency, Money]
> object Account { def apply(amounts: (Currency, Money)*): Account = amounts.toMap }

> val empty = Monoid[Account].empty
val empty: Account = Map()

> val account1 = Account(USD -> 10, GBP -> 5, EUR -> 1)
val account1: Account = Map(USD -> 10, GBP -> 5, EUR -> 1)

> val account2 = Account(GBP -> 2)
val account2: Account = Map(GBP -> 2)

> val account3 = Account(USD -> 3, EUR -> 5)
val account3: Account = Map(USD -> 3, EUR -> 5)

> account1 |+| empty
val res1: Account = Map(USD -> 10, GBP -> 5, EUR -> 1)

> account1 |+| account2
val res2: Account = Map(USD -> 10, GBP -> 7, EUR -> 1)

> List(account1, account2, account3).combineAll
val res3: Map[Currency, Money] = Map(GBP -> 7, USD -> 13, EUR -> 6)

> List(account1, account2, account3).foldMap(_.transform{ case (k,v) => v * 10 })
val res4: Map[Currency, Int] = Map(GBP -> 70, USD -> 130, EUR -> 60)
```

$\{A = Int \to Sum\ Int, \oplus = \lambda f.\lambda g.\lambda x.f(x) \oplus g(x), 1_A = \lambda x.0\}$
$\{A = Sum\ Int, \oplus = +, 1_A = 0\}$



$\{A = Int \Rightarrow Int, \oplus = \lambda f.\lambda g.\lambda x.f(x) \oplus g(x), 1_A = \lambda x.0\}$
$\{A = Int, \oplus = +, 1_A = 0\}$



```haskell
newtype Sum a = Sum {getSum :: a}
```

```haskell
instance Num a => Monoid (Sum a)
instance Monoid b => Monoid (a -> b)
```

```haskell
> increment n = n + 1
> twice n = 2 * n
> square n = n * n

> getSum ((mempty :: Int -> Sum Int) 33)
0

> getSum ((increment <> mempty) 5)
6

> getSum ((mempty <> twice) 5)
10

> getSum ((increment <> twice) 5)
16

> getSum ((mempty <> (mempty::Int -> Sum Int)) 5)
0

> getSum (fold ([]::[Int -> Sum Int]) 5)
0

> getSum (fold [increment, twice, square ] 5)
41
```

```scala
> import cats.implicits.*
> import cats.syntax.*
> import cats.Monoid
```

```scala
type Endo[A] = A => A
```

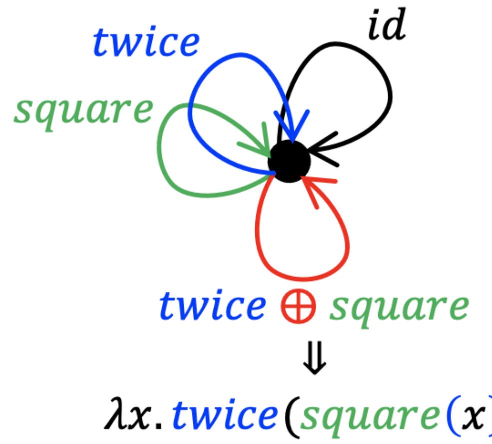
```scala
> import cats.Endo
```

```scala
> val empty = Monoid[Endo[Int]].empty
val empty: cats.Endo[Int] = ...

> def increment(n: Int): Int = n + 1
> def twice(n: Int): Int = 2 * n
> def square(n: Int): Int = n * n

> empty(33)
val res0: Int = 0

> (increment |+| empty)(5)
val res1: Int = 6

> (empty |+| twice)(5)
val res2: Int = 10

> (increment |+| twice)(5)
val res3: Int = 16

> (empty |+| empty)(5)
val res4: Int = 0

> List.empty[Endo[Int]].combineAll.apply(5)
val res5: Int = 0

> List(increment,twice,square).combineAll.apply(5)
val res6: Int = 41
```

$\{A = Int \rightarrow Int, \oplus = ., 1_A = id\}$

$(f\ .\ g)\ x = f\ (g\ x)$ – *composition function*

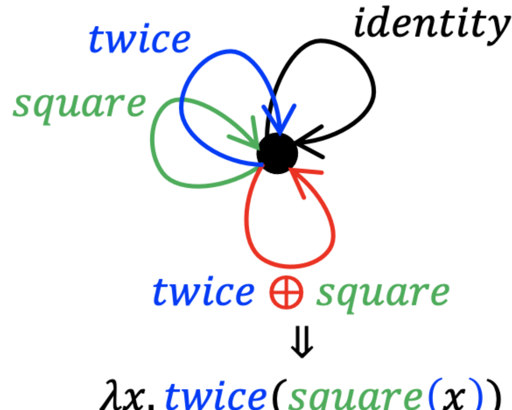
$id\ f\ x = f\ x$ – *identity function*



$\{A = Int \Rightarrow Int, \oplus = \text{compose}, 1_A = identity\}$

$(f\ compose\ g)(x) = f\ (g(x))$ – *composition function*

$(identity\ f)(x) = f(x)$ – *identity function*



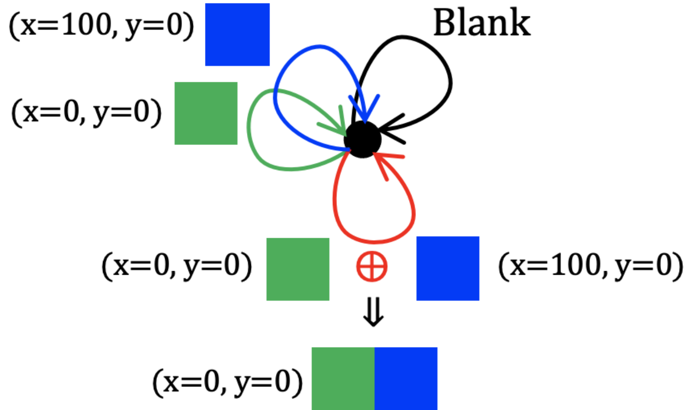
```haskell
type Endo :: * -> *
newtype Endo a = Endo {appEndo :: a -> a}
...
instance Monoid (Endo a)
instance Semigroup (Endo a)
```

```haskell
> increment n = n + 1
> twice n = 2 * n
> square n = n * n

> appEndo (mempty::Endo Int) 33
33

> appEndo ((Endo increment) <> (Endo twice)) 5
11

> appEndo (mempty <> (Endo twice)) 5
10

> appEndo (Endo increment <> mempty) 5
6

> appEndo (mempty <> (mempty::Endo Int)) 5
5

> appEndo (fold (fmap Endo [increment,twice,square]))5
51

> appEndo (foldMap Endo [increment,twice,square]) 5
51
```

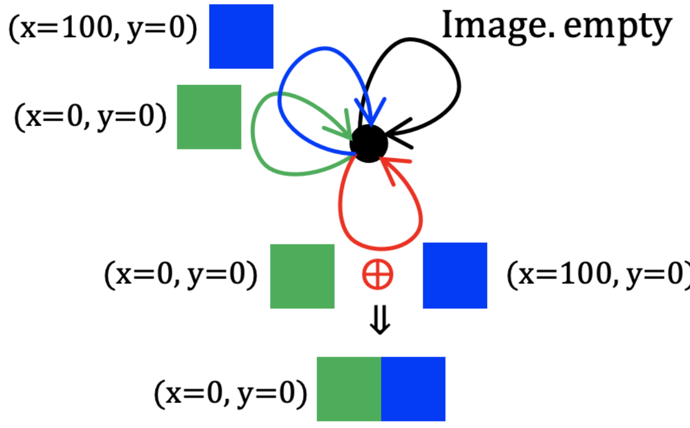
```scala
import cats.implicits.catsSyntaxSemigroup
import cats.syntax.foldable.*
import cats.{Monoid,MonoidK}

type Endo[A] = A=>A
import cats.Endo
```

```scala
> given endoMonoid: Monoid[Int => Int] =
    MonoidK[Endo].algebra[Int]

> def increment(n: Int): Int = n + 1
> def twice(n: Int): Int = 2 * n
> def square(n: Int): Int = n * n

> empty(33)
val res0: Int = 33

> (increment |+| twice)(5)
val res1: Int = 11

> (endoMonoid.empty |+| twice)(5)
val res2: Int = 10

> (increment |+| endoMonoid.empty)(5)
val res3: Int = 6

> (endoMonoid.empty |+| endoMonoid.empty)(5)
val res4: Int = 5

> List.empty[Int => Int].combineAll.apply(5)
val res5: Int = 5

> List(increment,twice,square).combineAll.apply(5)
val res6: Int = 51
```

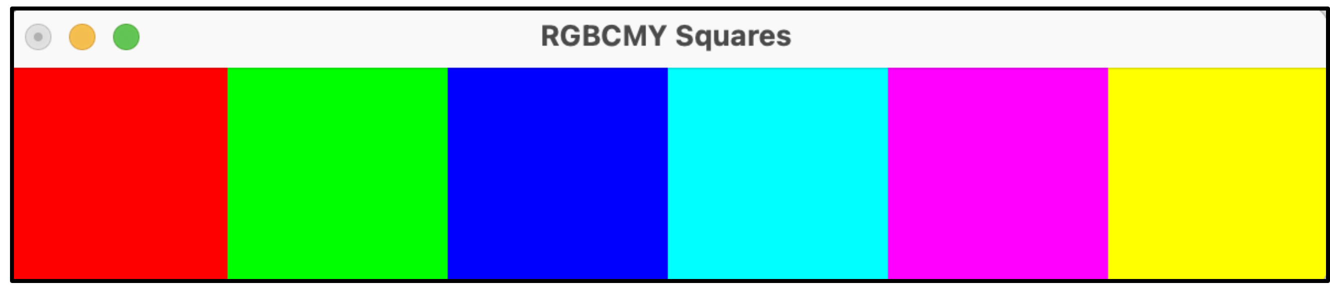
$\{A = \text{Picture}, \oplus = \text{combine superimposing}, 1_A = \text{Picture.Blank}\}$



$\{A = \text{Image}, \oplus = \text{combine superimposing}, 1_A = \text{Image.empty}\}$

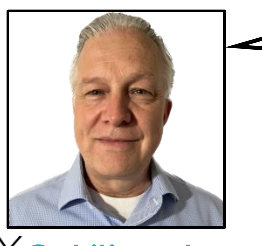


```haskell
import Graphics.Gloss

main :: IO ()
main = display window white rgbcmyRectangle

squareImage = rectangleSolid (fromIntegral 100) (fromIntegral 100)

[redSquare, greenSquare, blueSquare, cyanSquare, magentaSquare, yellowSquare] =
  fmap (\(colour, index) -> translate (-100 * (2-index) - 50) 0 (color colour squareImage))
       (zip [red, green, blue, cyan, magenta, yellow] [0..])

rgbRectangle = redSquare <> greenSquare <> blueSquare

rgbcmyRectangle = fold [rgbRectangle, cyanSquare, magentaSquare, yellowSquare]

window = InWindow "RGBCMY Squares" (600, 100) (0,0)
```



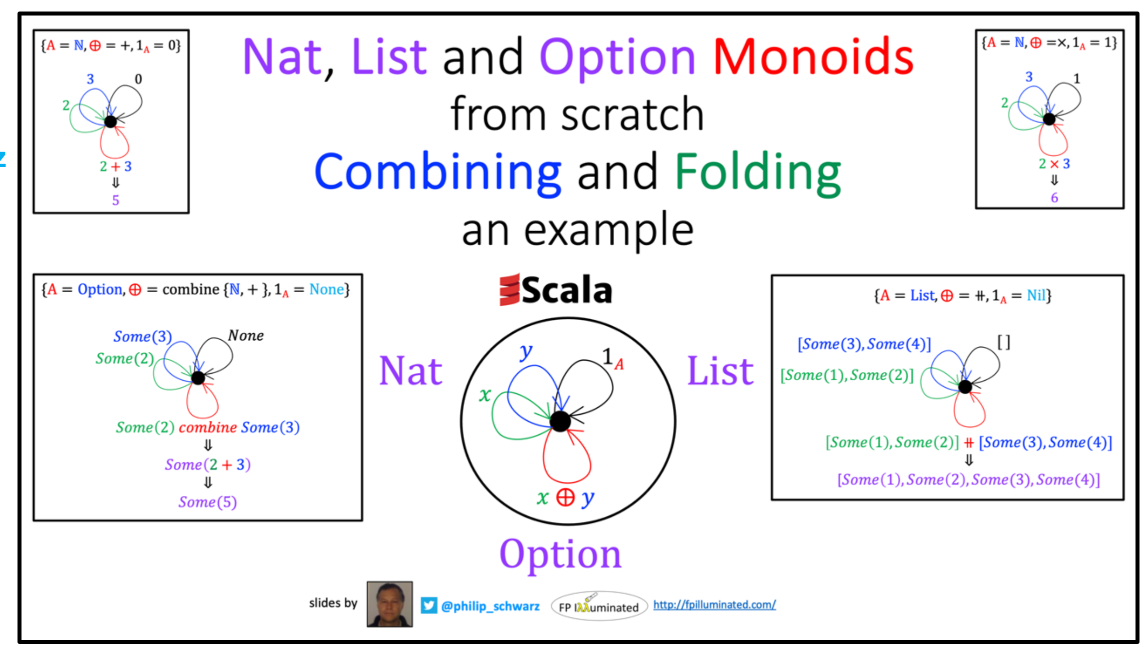
```scala
import cats.Monoid
import cats.effect.unsafe.implicits.global
import cats.implicits.*
import doodle.core.*
import doodle.image.*
import doodle.image.syntax.*
import doodle.image.syntax.all.*
import doodle.image.syntax.core.*
import doodle.java2d.*

@main
def main(): Unit =

  val squareImage = Image.square(100)

  val List(redSquare, greenSquare, blueSquare, cyanSquare, magentaSquare, yellowSquare) =
    List(Color.red, Color.green, Color.blue, Color.cyan, Color.magenta, Color.yellow)
      .zipWithIndex.map((color, index) => squareImage.at(index * 100, 0).fillColor(color))

  given Monoid[Image] = Monoid.instance[Image](Image.empty, _ on _)

  val rgbRectangle: Image = redSquare |+| greenSquare |+| blueSquare

  val rgbcmyRectangle: Image = List(rgbRectangle, cyanSquare, magentaSquare, yellowSquare).combineAll

  rgbcmyRectangle.drawWithFrame(Frame.default.withTitle("RGBCMY Squares"))
```

RGBCMY Squares



@philip_schwarz

If you liked this deck, you may also be interested in one or more of the following



https://fpilluminated.com/