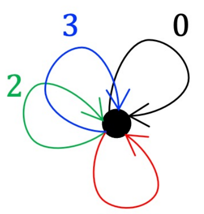
Nat, List and Option Monoids from scratch

Combining and Folding

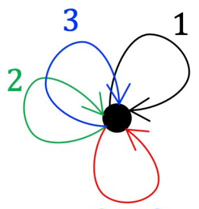
an example

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



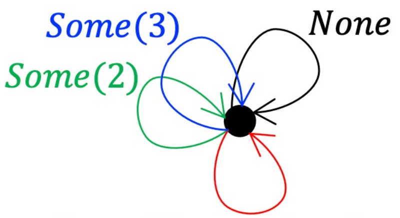
$2 + 3$
$\Downarrow$
5

$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$



2×3
$\Downarrow$
6

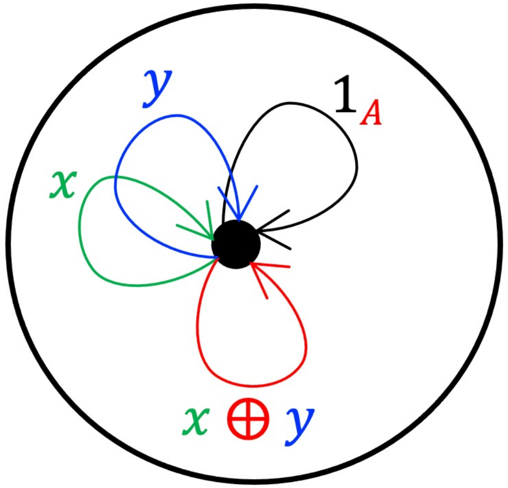
$\{A = \text{Option}, \oplus = \text{combine}\ \{\mathbb{N}, +\}, 1_A = \text{None}\}$



$Some(2)\ combine\ Some(3)$
$\Downarrow$
$Some(2 + 3)$
$\Downarrow$
$Some(5)$

Scala

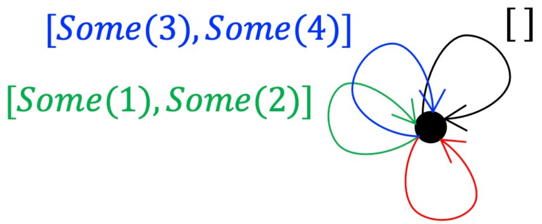
Nat



List

Option

$\{A = \text{List}, \oplus = +\!\!+, 1_A = \text{Nil}\}$



$[Some(1), Some(2)]\ +\!\!+\ [Some(3), Some(4)]$
$\Downarrow$
$[Some(1), Some(2), Some(3), Some(4)]$

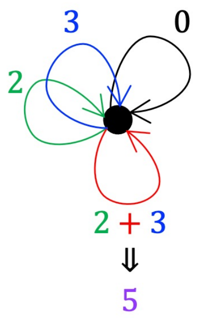
slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

```haskell
data Nat = Zero | Succ Nat
```

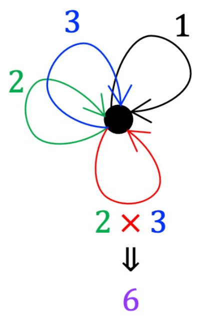
```haskell
(+)           :: Nat → Nat → Nat
m + Zero      =  m
m + Succ n    =  Succ (m + n)
```

```haskell
(×)           :: Nat → Nat → Nat
m × Zero      =  Zero
m × Succ n    =  (m × n) + m
```

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



$\{A = \mathbb{N}, \oplus = \times, 1_A = 1\}$

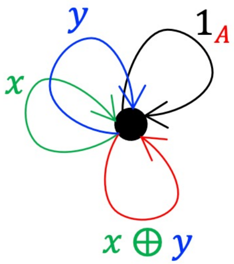


Monoid

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: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

```scala
enum Nat:
  case Zero
  case Succ(n: Nat)
```

```scala
extension (m: Nat)

  def +(n: Nat): Nat = n match
    case Zero => m
    case Succ(n) => Succ(m + n)

  def *(n: Nat): Nat = n match
    case Zero => Zero
    case Succ(n) => m * n + m
```

```scala
val zero = Zero
val one = Succ(zero)
val two = Succ(one)

val three = Succ(two)
val four = Succ(three)
val five = Succ(four)
val six = Succ(five)
```

```scala
assert( zero + one == one )
assert( one + zero == one )

assert( one + two == three )
assert( one + two + three == six )
```

```scala
assert( two * one == two )
assert( one * two == two )

assert( two * three == six )
assert( one * two * three == six )
```

```scala
trait Semigroup[A]:

  def combine(x: A, y: A): A

object Semigroup:

  extension [A](lhs: A)(using m: Semigroup[A])

    def ⊕(rhs: A): A = m.combine(lhs,rhs)
```

```scala
trait Monoid[A] extends Semigroup[A]:

  def unit: A
```

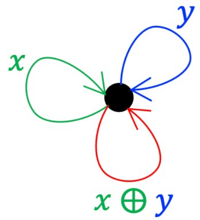
```scala
given Monoid[Nat] with
  def unit: Nat = Zero
  def combine(x: Nat, y: Nat): Nat = x + y
```

```scala
assert( summon[Monoid[Nat]].combine(two,three) == five )

assert( (two ⊕ three) == five )

assert( (one ⊕ two ⊕ three) == six )
```

Semigroup

A: type (set of values)
$\oplus$: $A \times A \longrightarrow A$

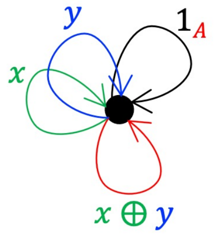
Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



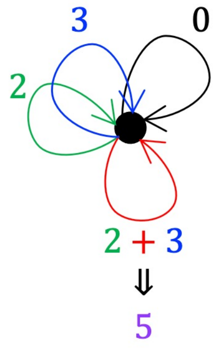
Monoid

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: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$



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



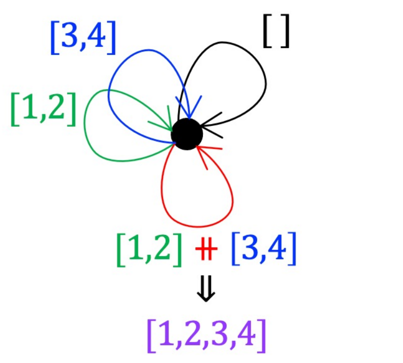
```haskell
data List α = Nil | Cons α (List α)
```

```haskell
Cons 1 (Cons 2 (Cons 3 Nil))
```

```scala
enum List[+A]:
  case Cons(head: A, tail: List[A])
  case Nil
```

```scala
object List:

  def append[A](lhs: List[A], rhs: List[A]): List[A] = lhs match
    case Nil => rhs
    case Cons(a, rest) => Cons(a,append(rest,rhs))

  extension [A](lhs: List[A])
    def ++(rhs: List[A]): List[A] = append(lhs,rhs)
```

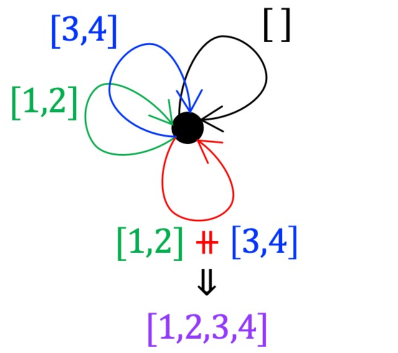
$\{A = \text{List}, \oplus = +\!\!+, 1_A = \text{Nil}\}$




```scala
val oneTwo = Cons(one,Cons(two,Nil))
val threeFour = Cons(three,Cons(four,Nil))

assert(append(oneTwo,threeFour) == Cons(one,Cons(two,Cons(three,Cons(four,Nil)))))
assert(oneTwo ++ threeFour == Cons(one,Cons(two,Cons(three,Cons(four,Nil)))))
```

```
object List:

  def apply[A](as: A*): List[A] = as match
    case Seq() => Nil
    case _ => Cons(as.head, List(as.tail*))

  def append[A](lhs: List[A], rhs: List[A]): List[A] = lhs match
    case Nil => rhs
    case Cons(a, rest) => Cons(a,append(rest,rhs))

  extension [A](lhs: List[A])
    def ++(rhs: List[A]): List[A] = append(lhs,rhs)
```

$\{A = \text{List}, \oplus = ++, 1_A = \text{Nil}\}$



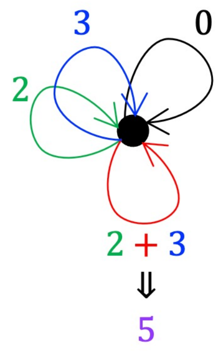

```
assert(List(one,two,three) == Cons(one,Cons(two,Cons(three,Nil))))
assert(List(one,two) ++ List(three, four) ++ Nil == List(one,two,three,four))
```

```
given ListMonoid[A]: Monoid[List[A]] with
  def unit: List[A] = Nil
  def combine(lhs: List[A], rhs: List[A]): List[A] = lhs ++ rhs
```

```
assert(summon[Monoid[List[Nat]]].combine(List(one,two),List(three, four)) == List(one,two,three,four))
assert((List(one,two) ⊕ List(three, four)) == List(one,two,three,four))
```

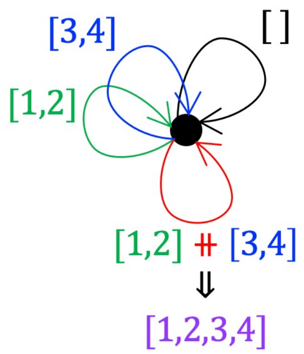
```scala
object List:

  def apply[A](as: A*): List[A] = as match
    case Seq() => Nil
    case _ => Cons(as.head, List(as.tail*))

  def nil[A]: List[A] = Nil

  def append[A](lhs: List[A], rhs: List[A]): List[A] = lhs match
    case Nil => rhs
    case Cons(a, rest) => Cons(a,append(rest,rhs))

  extension [A](lhs: List[A])
    def ++(rhs: List[A]): List[A] = append(lhs,rhs)

  def fold[A](as: List[A])(using ma: Monoid[A]): A = as match
    case Nil => ma.unit
    case Cons(a,rest) => ma.combine(a,fold(rest))
```

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




```scala
assert(fold(List(one,two,three,four)) == one + two + three + four)
assert(fold(nil[Nat]) == zero)

assert(fold(List(List(one,two),Nil,List(three, four),List(five,six)))
      == List(one,two,three,four,five,six))
```

$\{A = \text{List}, \oplus = +\!\!+, 1_A = \text{Nil}\}$

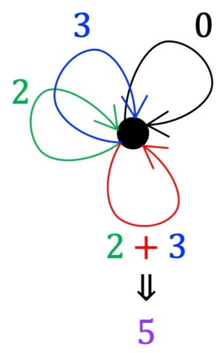


```scala
object List:

  def apply[A](as: A*): List[A] = as match
    case Seq() => Nil
    case _ => Cons(as.head, List(as.tail*))

  def nil[A]: List[A] = Nil

  def append[A](lhs: List[A], rhs: List[A]): List[A] = lhs match
    case Nil => rhs
    case Cons(a, rest) => Cons(a,append(rest,rhs))

  extension [A](lhs: List[A])
    def ++(rhs: List[A]): List[A] = append(lhs,rhs)

  def fold[A](as: List[A])(using ma: Monoid[A]): A =
    foldRight(as, ma.unit, (a,b) => ma.combine(a,b))

  def foldRight[A,B](as: List[A], b: B, f: (A, B) => B): B = as match
    case Nil => b
    case Cons(a,rest) => f(a,foldRight(rest,b,f))
```

Same as the previous slide, except that here we define **fold** in terms of **foldRight**.

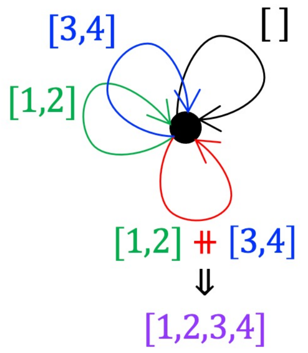
@philip_schwarz

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



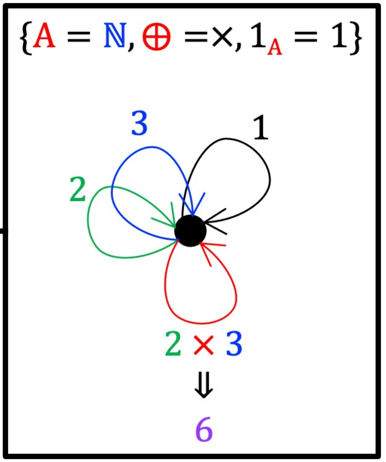

```scala
assert(fold(List(one,two,three,four)) == one + two + three + four)
assert(fold(nil[Nat]) == zero)

assert(fold(List(List(one,two),Nil,List(three, four),List(five,six)))
      == List(one,two,three,four,five,six))
```

$\{A = List, \oplus = +\!\!+, 1_A = Nil\}$



```
val natMultMonoid = new Monoid[Nat]:
  def unit: Nat = Succ(Zero)
  def combine(x: Nat, y: Nat): Nat = x * y
```

```
assert(fold(List(one,two,three,four))(using natMultMonoid) == one * two * three * four)
assert(fold(nil[Nat])(using natMultMonoid) == one)
```



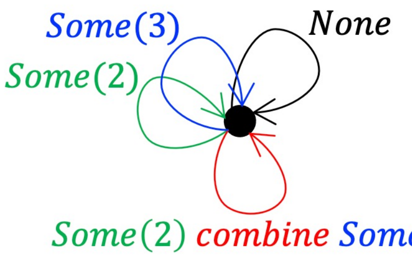
data Maybe α = Nothing | Just α

```
enum Option[+A]:
  case None
  case Some(value:A)
```

```
object Option:
  def none[A]: Option[A] = None
  def some[A](a:A): Option[A] = Some(a)
```

```
given OptionMonoid[A:Semigroup]: Monoid[Option[A]] with
  def unit: Option[A] = None
  def combine(ox: Option[A], oy: Option[A]): Option[A] = (ox,oy) match
    case (None,_) => oy
    case (_,None) => ox
    case (Some(x),Some(y)) => Some(x ⊕ y)
```

$\{A = \text{Option}, \oplus = \text{combine}\ \{\mathbb{N}, +\}, 1_A = \text{None}\}$



$\Downarrow$

$Some(2+3)$

$\Downarrow$

$Some(5)$

```
assert(summon[Monoid[Option[Nat]]].combine(Some(two),Some(three)) == Some(five))
assert(summon[Monoid[Option[Nat]]].combine(Some(two),None) == Some(two))
assert(summon[Monoid[Option[Nat]]].combine(none[Nat],Some(two)) == Some(two))
assert(summon[Monoid[Option[Nat]]].combine(none[Nat],None) == None)
```

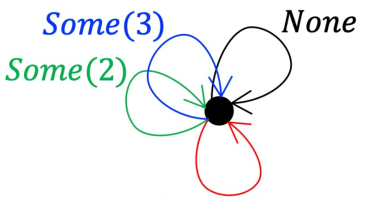
```
assert((some(two) ⊕ None) == Some(two))
assert((none[Nat] ⊕ Some(two)) == Some(two))
assert((some(two) ⊕ Some(three)) == Some(five))
assert((none[Nat] ⊕ None) == None)
```

```scala
assert(fold(List(Some(two),None,Some(three))) == Some(five))
assert(fold(nil[Option[Nat]]) == None)
```

```scala
assert((List(Some(one),None,Some(two)) ++ List(Some(three),None,Some(four)))
        == List(Some(one),None,Some(two),Some(three),None,Some(four)))

assert(summon[Monoid[List[Option[Nat]]]].combine(List(Some(one),None,Some(two)),List(Some(three),None,Some(four)))
       == List(Some(one),None,Some(two),Some(three),None,Some(four)))

assert((List(Some(one),None,Some(two)) ⊕ List(Some(three),None,Some(four)))
       == List(Some(one),None,Some(two),Some(three),None,Some(four)))
```

```scala
assert(fold(List(Some(one),None,Some(two)) ⊕ List(Some(three),None,Some(four)))
       == Some(one + two + three + four))
```

$\{A = \text{Option}, \oplus = \text{combine}\ \{\mathbb{N}, +\}, 1_A = \text{None}\}$



$Some(2)\ combine\ Some(3)$

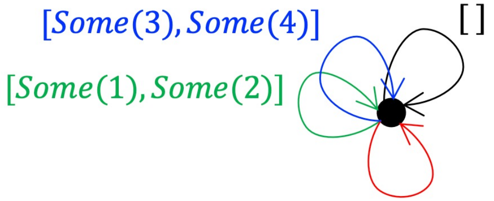
$\Downarrow$

$Some(2 + 3)$

$\Downarrow$

$Some(5)$

$\{A = \text{List}, \oplus = +\!\!+, 1_A = \text{Nil}\}$



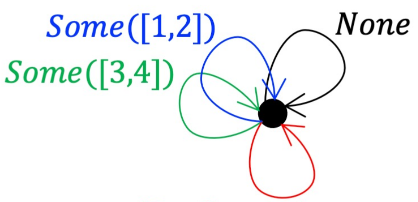
$[Some(1), Some(2)] +\!\!+ [Some(3), Some(4)]$

$\Downarrow$

$[Some(1), Some(2), Some(3), Some(4)]$

```scala
assert((some(List(one,two)) ⊕ None ⊕ Some(List(three,four)))
      == Some(List(one,two,three,four)))
```

$\{A = \text{Option}, \oplus = \text{combine}\,\{\text{List}, +\!\!+\}, 1_A = \text{None}\}$



$Some([1,2])\ combine\ Some([3,4])$

$\Downarrow$

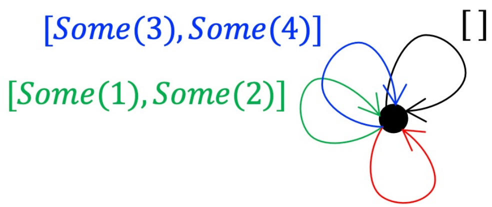
$Some([1,2] +\!\!+ [3,4])$

$\Downarrow$

$Some([1,2,3,4])$

```scala
assert(
  fold(
    fold(
      List(List(Some(one), None, Some(two)),
           List(Some(three), None, Some(four)),
           List(Some(five), None, Some(six)))
    )
  )
  == Some(one + two + three + four + five + six))
```

$\{A = \text{List}, \oplus = +\!\!+, 1_A = \text{Nil}\}$

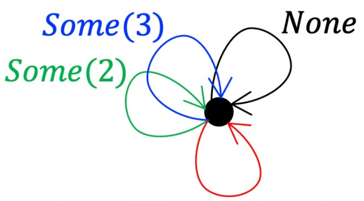


$[Some(1), Some(2)] +\!\!+ [Some(3), Some(4)]$

$\Downarrow$

$[Some(1), Some(2), Some(3), Some(4)]$

$\{A = \text{Option}, \oplus = \text{combine}\,\{\mathbb{N}, +\}, 1_A = \text{None}\}$



$Some(2)\ combine\ Some(3)$

$\Downarrow$

$Some(2 + 3)$

$\Downarrow$

$Some(5)$

That’s all.

I hope you found it useful.

@philip_schwarz