Folding

```
      :                                   f
     / \                                 / \
   a0   :                              a0   f
       / \                                 / \
     a1   :                              a1   f
         / \                                 / \
       a2   :                              a2   f
           / \                                 / \
         a3   []                             a3   e
```

CHEAT-SHEET

#7

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

The Three **Duality Theorems** of **Fold**

(for all finite lists ☩)

① $foldr\ (\oplus)\ e\ xs\ = foldl\ (\oplus)\ e\ xs$

where $\oplus$ and e are such that 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. ✠

†

② $foldr\ (\oplus)\ e\ xs\ = foldl\ (\otimes)\ e\ xs$

where $\oplus$, $\otimes$, and e are such that for all x, y, and z we have

$$x \oplus (y \otimes z)\ = (x \oplus y) \otimes z$$
$$x \oplus e\ = e \otimes x$$

In other words, $\oplus$ and $\otimes$ **associate** with each other, and e on the right of $\oplus$ is equivalent to e on the left of $\otimes$.

‡

③ $foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$

where $flip\ f\ x\ y = f\ y\ x$

$$\begin{aligned} &foldr\ ::\ (\alpha\ \to \beta\ \to \beta) \to \beta \to [\alpha] \to \beta \\ &foldr\ f\ e\ [\,] \qquad\ = e \\ &foldr\ f\ e\ (x{:}xs) = f\ x\ (foldr\ f\ e\ xs) \end{aligned}$$

$$\begin{aligned} &foldl\ ::\ (\beta\ \to \alpha\ \to \beta) \to \beta \to [\alpha] \to \beta \\ &foldl\ f\ e\ [\,] \qquad\ = e \\ &foldl\ f\ e\ (x{:}xs) = foldl\ f\ (f\ e\ x)\ xs \end{aligned}$$

† Theorem ① is a special case of ② with $(\oplus) = (\otimes)$

‡ Theorem ② is a generalisation of ①

✠ For example, $+$ and $\times$ are **associative** operators with respective **units** 0 and 1.

☩ Except lists sufficiently large to cause a **right fold** to encounter a **stack overflow**

1 $foldr\ (\oplus)\ e\ xs\ = foldl\ (\oplus)\ e\ xs$

where $\oplus$ and e are such that 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.

associative operator	unit
+	0
*	1
&&	True
||	False
++	[]

```
:{
foldRight :: (α -> β -> β) -> β -> [α] -> β
foldRight f e []     = e
foldRight f e (x:xs) = f x (foldRight f e xs)

foldLeft :: (β -> α -> β) -> β -> [α] -> β
foldLeft f e []     = e
foldLeft f e (x:xs) = foldLeft f (f e x) xs
:}

> sumLeft   =  foldLeft (+) 0
> sumRight = foldRight (+) 0

> subLeft   =  foldLeft (-) 0
> subRight = foldRight (-) 0

> prdLeft   =  foldLeft (*) 1
> prdRight = foldRight (*) 1

> andLeft   =  foldLeft (&&) True
> andRight = foldRight (&&) True

> orLeft    =  foldLeft (||) False
> orRight   = foldRight (||) False

> concatLeft   =  foldLeft (++) []
> concatRight = foldRight (++) []
```

```
> integers = [1,2,3,4]
> flags = [True, False, True]
> lists = [[1], [2,3,4],[5,6]]

> subLeft(integers)
-10

> subRight(integers)
-2
```

subtraction is not **associative**, and **0** is not its **unit**, so the following are not equivalent:

```
foldLeft (-) 0
foldRight (-) 0
```

```
> assert (sumLeft(integers) == sumRight(integers)) "OK"
"OK"

> assert (subLeft(integers) /= subRight(integers)) "OK"
"OK"

> assert (prdLeft(integers) == prdRight(integers)) "OK"
"OK"

> assert (andLeft(flags) == andRight(flags)) "OK"
"OK"

> assert (orLeft(flags) == orRight(flags)) "OK"
"OK"

> assert (concatLeft(lists) == concatRight(lists)) "OK"
"OK"
```

Same as previous slide but using built-in foldl and foldr

```
> sumLeft   = foldl (+) 0
> sumRight = foldr (+) 0

> subLeft   = foldl (-) 0
> subRight = foldr (-) 0

> prdLeft   = foldl (*) 1
> prdRight = foldr (*) 1

> andLeft   = foldl (&&) True
> andRight = foldr (&&) True

> orLeft    = foldl (||) False
> orRight   = foldr (||) False

> concatLeft  = foldl (++) []
> concatRight = foldr (++) []
```

```
> integers = [1,2,3,4]
> flags = [True, False, True]
> lists = [[1], [2,3,4],[5,6]]

> subLeft(integers)
-10

> subRight(integers)
-2
```

subtraction is not **associative**, and **0** is not its **unit**, so the following are not equivalent:

```
foldl (-) 0
foldr (-) 0
```

```
> assert (sumLeft(integers) == sumRight(integers)) "OK"
"OK"

> assert (subLeft(integers) /= subRight(integers)) "OK"
"OK"

> assert (prdLeft(integers) == prdRight(integers)) "OK"
"OK"

> assert (andLeft(flags) == andRight(flags)) "OK"
"OK"

> assert (orLeft(flags) == orRight(flags)) "OK"
"OK"

> assert (concatLeft(lists) == concatRight(lists)) "OK"
"OK"
```

```scala
def foldr[A, B](f: A => B => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => f(x)(foldr(f)(e)(xs))

def foldl[A, B](f: B => A => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => foldl(f)(f(e)(x))(xs)

val `(+)`: Int => Int => Int = m => n => m + n
val `(-)`: Int => Int => Int = m => n => m - n
val `(*)`: Int => Int => Int = m => n => m * n
val `(&&)`: Boolean => Boolean => Boolean = m => n => m && n
val `(||)`: Boolean => Boolean => Boolean = m => n => m || n
def `(++)`[A](m: Seq[A]): Seq[A] => Seq[A] = n => m ++ n

val sumLeft =  foldl(`(+)`)(0)
val sumRight = foldr(`(+)`)(0)

val subLeft =  foldl(`(-)`)(0)
val subRight = foldr(`(-)`)(0)

val prodLeft =  foldl(`(*)`)(1)
val prodRight = foldr(`(*)`)(1)

val andLeft =  foldl(`(&&)`)(true)
val andRight = foldr(`(&&)`)(true)

val orLeft =  foldl(`(||)`)(true)
val orRight = foldr(`(||)`)(true)

val concatLeft =  foldl(`(++)`)(Nil)
val concatRight = foldr(`(++)`)(Nil)
```

associative operator	unit
+	0
*	1
&&	True
||	False
++	[]

1 $foldr\ (\oplus)\ e\ xs\ =\ foldl\ (\oplus)\ e\ xs$

where $\oplus$ and e are such that for all x, y, and z we have

$(x \oplus y) \oplus z = x \oplus (y \oplus z)$
$e \oplus x = x$ and $x \oplus e = x$

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

```scala
val integers = List(1, 2, 3, 4)
val flags    = List(true, false, true)
val lists    = List(List(1), List(2, 3, 4), List(5, 6))

scala> subLeft(integers)
val res0: Int = -10

scala> subRight(integers)
val res1: Int = -2

scala>   assert(  sumLeft(integers) == sumRight(integers)   )
     |   assert(  subLeft(integers) != subRight(integers)   )
     |   assert( prodLeft(integers) == prodRight(integers) )
     |   assert(     andLeft(flags) == andRight(flags)      )
     |   assert(      orLeft(flags) == orRight(flags)       )
     |   assert(  concatLeft(lists) == concatRight(lists)  )
scala>
```

subtraction is not **associative**, and **0** is not its **unit**, so the following are not equivalent:

```scala
foldl(`(-)`)(0)
foldr(`(-)`)(0)
```

Same as previous slide but using built-in foldLeft and foldRight

```scala
val sumLeft:  List[Int] => Int = _.foldLeft(0)(_+_)
val sumRight: List[Int] => Int = _.foldRight(0)(_+_)

val subLeft:  List[Int] => Int = _.foldLeft(0)(_-_)
val subRight: List[Int] => Int = _.foldRight(0)(_-_)

val prodLeft:  List[Int] => Int = _.foldLeft(1)(_*_)
val prodRight: List[Int] => Int = _.foldRight(1)(_*_)

val andLeft:  List[Boolean] => Boolean = _.foldLeft(true)(_&&_)
val andRight: List[Boolean] => Boolean = _.foldRight(true)(_&&_)

val orLeft:  List[Boolean] => Boolean = _.foldLeft(false)(_||_)
val orRight: List[Boolean] => Boolean = _.foldRight(false)(_||_)

def concatLeft[A]: List[List[A]] => List[A] =
  _.foldLeft(List.empty[A])(_++_)
def concatRight[A]: List[List[A]] => List[A] =
  _.foldRight(List.empty[A])(_++_)
```

```scala
val integers = List(1, 2, 3, 4)
val flags = List(true, false, true)
val lists = List(List(1), List(2, 3, 4), List(5, 6))

scala> subLeft(integers)
val res0: Int = -10

scala> subRight(integers)
val res1: Int = -2

scala>   assert(  sumLeft(integers) == sumRight(integers)  )
     |   assert(  subLeft(integers) != subRight(integers)  )
     |   assert( prodLeft(integers) == prodRight(integers) )
     |   assert(     andLeft(flags) == andRight(flags)     )
     |   assert(      orLeft(flags) == orRight(flags)      )
     |   assert(  concatLeft(lists) == concatRight(lists)  )
scala>
```

subtraction is not **associative**, and **0** is not its **unit**, so the following are not equivalent:

```scala
_.foldLeft(0)(_-_)
_.foldRight(0)(_-_)
```

② $foldr\ (\oplus)\ e\ xs\ = foldl\ (\otimes)\ e\ xs$

Haskell

where $\oplus$, $\otimes$, and e are such that for all x, y, and z we have

$$x \oplus (y \otimes z) = (x \oplus y) \otimes z$$
$$x \oplus e = e \otimes x$$

In other words, $\oplus$ and $\otimes$ **associate** with each other, and e on the right of $\oplus$ is equivalent to e on the left of $\otimes$.

```
:{
foldRight :: (α -> β -> β) -> β -> [α] -> β
foldRight f e []     = e
foldRight f e (x:xs) = f x (foldRight f e xs)

foldLeft :: (β -> α -> β) -> β -> [α] -> β
foldLeft f e []     = e
foldLeft f e (x:xs) = foldLeft f (f e x) xs
:}
```

```
list = [1,2,3]
```

```
> lengthRight = foldRight oneplus 0 where oneplus x n = 1 + n
> lengthLeft  = foldLeft  plusone 0 where plusone n x = n + 1

> assert (lengthRight(list) == lengthLeft(list)) "OK"
"OK”
```

```
> reverseRight = foldRight snoc [] where snoc x xs = xs ++ [x]
> reverseLeft  = foldLeft  cons [] where cons xs x = x : xs

> assert (reverseRight(list) == reverseLeft(list)) "OK"
"OK"
```

Same as on the left but using built-in foldl and foldr

```
> lengthRight = foldr oneplus 0 where oneplus x n = 1 + n
> lengthLeft  = foldl plusone 0 where plusone n x = n + 1

> assert (lengthRight(list) == lengthLeft(list)) "OK"
"OK”
```

```
> reverseRight = foldr snoc [] where snoc x xs = xs ++ [x]
> reverseLeft  = foldl cons [] where cons xs x = x : xs

> assert (reverseRight(list) == reverseLeft(list)) "OK"
"OK"
```

2 $foldr\ (\oplus)\ e\ xs\ = foldl\ (\otimes)\ e\ xs$

Scala

where $\oplus$, $\otimes$, and e are such that for all x, y, and z we have

$$x \oplus (y \otimes z) = (x \oplus y) \otimes z$$
$$x \oplus e = e \otimes x$$

In other words, $\oplus$ and $\otimes$ **associate** with each other, and e on the right of $\oplus$ is equivalent to e on the left of $\otimes$.

```
def foldr[A, B](f: A => B => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => f(x)(foldr(f)(e)(xs))

def foldl[A, B](f: B => A => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => foldl(f)(f(e)(x))(xs)
```

```
val list: List[Int] = List(1, 2, 3)
```

```
def oneplus[A]: A => Int => Int = x => n => 1 + n
def plusOne[A]: Int => A => Int = n => x => n + 1

val lengthRight = foldr(oneplus)(0)
val lengthLeft  = foldl(plusOne)(0)

scala> assert( lengthRight(list) == lengthLeft(list) )
```

```
def snoc[A]: A => List[A] => List[A] = x => xs => xs ++ List(x)
def cons[A]: List[A] => A => List[A] = xs => x => x::xs

val reverseRight = foldr(snoc[Int])(Nil)
val reverseLeft  = foldl(cons[Int])(Nil)

scala> assert( reverseRight(list) == reverseLeft(list) )
```

Same as on the left but using built-in foldLeft and foldRight

```
def oneplus[A]: (A, Int) => Int = (x, n) => 1 + n
def plusOne[A]: (Int, A) => Int = (n, x) => n + 1

def lengthRight[A]: List[A] => Int = _.foldRight(0)(oneplus)
def lengthLeft[A]:  List[A] => Int = _.foldLeft(0)(plusOne)

scala> assert( lengthRight(list) == lengthLeft(list) )
```

```
def snoc[A]:(A, List[A]) => List[A] = (x, xs) => xs ++ List(x)
def cons[A]:(List[A], A) => List[A] = (xs, x) => x::xs

def reverseRight[A]: List[A]=>List[A] = _.foldRight(Nil)(snoc)
def reverseLeft[A] : List[A]=>List[A] = _.foldLeft(Nil)(cons)

scala> assert( reverseRight(list) == reverseLeft(list) )
```

③ $foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$ (Also holds true when $foldr$ and $foldl$ are swapped)

Haskell

```
:{
foldRight :: (α -> β -> β) -> β -> [α] -> β
foldRight f e []     = e
foldRight f e (x:xs) = f x (foldRight f e xs)

foldLeft :: (β -> α -> β) -> β -> [α] -> β
foldLeft f e []     = e
foldLeft f e (x:xs) = foldLeft f (f e x) xs
:}
```

```
> sumRight = foldRight (+) 0
> sumLeft  = foldLeft (flip (+)) 0 . reverse

> assert (sumRight(list) == sumLeft(list)) "OK"
"OK"
```

```
> oneplus x n = 1 + n
> lengthRight = foldRight oneplus 0
> lengthLeft = foldLeft (flip oneplus) 0 . reverse

> assert (lengthRight(list) == lengthLeft(list)) "OK"
"OK"
```

```
> n ⊕ d = 10 * n + d                                    †
> decimalLeft = foldLeft (⊕) 0
> decimalRight = foldRight (flip (⊕)) 0 . reverse

> assert (decimalLeft(list) == decimalRight(list))
"OK"
```

Same as on the left but using built-in foldl and foldr

```
> sumRight = foldr (+) 0
> sumLeft  = foldl (flip (+)) 0 . reverse

> assert (sumRight(list) == sumLeft(list)) "OK"
"OK"
```

```
> oneplus x n = 1 + n
> lengthRight = foldr oneplus 0
> lengthLeft = foldl (flip oneplus) 0 . reverse

> assert (lengthRight(list) == lengthLeft(list)) "OK"
"OK"
```

```
> n ⊕ d = 10 * n + d                                    †
> decimalLeft = foldl (⊕) 0
> decimalRight = foldr (flip (⊕)) 0 . reverse

> assert (decimalLeft(list) == decimalRight(list)) "OK"
"OK"
```

† see next slide

@philip_schwarz

At the bottom of the previous slide and the next one, instead of exploiting this equation

$$foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$$

we are exploiting the following derived equation in which $foldr$ is renamed to $foldl$ and vice versa:

$$foldl\ f\ e\ xs\ = foldr\ (flip\ f)\ e\ (reverse\ xs)$$

The equation can be derived as shown below.

Define $g = flip\ f$ and $ys = reverse\ xs$, from which it follows that $f = flip\ g$ and $xs = reverse\ ys$.

In the original equation, replace f with $(flip\ g)$ and replace xs with $(reverse\ ys)$

$$foldr\ (flip\ g)\ e\ (reverse\ ys) = foldl\ \big(flip(flip\ g)\big)\ e\ (reverse\ (reverse\ ys))$$

Simplify by replacing $flip(flip\ g)$ with g and $(reverse\ (reverse\ ys))$ with ys

$$foldr\ (flip\ g)\ e\ (reverse\ ys) = foldl\ g\ e\ ys$$

Swap the right hand side with left hand side

$$foldl\ g\ e\ ys = foldr\ (flip\ g)\ e\ (reverse\ ys)$$

Rename g to f and rename ys to xs

$$foldl\ f\ e\ xs = foldr\ (flip\ f)\ e\ (reverse\ xs)$$

3

$foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$

```
def foldr[A, B](f: A => B => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => f(x)(foldr(f)(e)(xs))

def foldl[A, B](f: B => A => B)(e: B)(s: List[A]): B = s match
  case Nil => e
  case x :: xs => foldl(f)(f(e)(x))(xs)
```

```
def flip[A, B, C]: (A => B => C) => (B => A => C) =
  f => b => a => f(a)(b)

val list: List[Int] = List(1, 2, 3)
```

```
def plus: Int => Int => Int = m => n => m + n

val sumRight = foldr(plus)(0)
val sumLeft  = (xs: List[Int]) => foldl(flip(plus))(0)(xs.reverse)

assert( sumRight(list) == sumLeft(list) )
```

```
def oneplus[A]: A => Int => Int = x => n => 1 + n

val lengthRight = foldr(oneplus)(0)
def lengthLeft[A] = (xs: List[A]) => foldl(flip(oneplus))(0)(xs.reverse)

assert( lengthRight(list) == lengthLeft(list) )
```

```
def `(⊕)`: Int => Int => Int = n => d => 10 * n + d

val decimalLeft = foldl(`(⊕)`)(0)
val decimalRight  = (xs: List[Int]) => foldr(flip(`(⊕)`))(0)(xs.reverse)

assert( decimalLeft(list) == decimalRight(list) )
```
†

† see previous slide

Same as previous slide but using built-in foldLeft and foldRight

```scala
def flip[A, B, C]: ((A,B) => C) => ((B,A) => C) = f => (b,a) => f(a,b)

val list: List[Int] = List(1, 2, 3)
```

```scala
def plus: (Int,Int) => Int = (m,n) => m + n

val sumRight: List[Int] => Int = _.foldRight(0)(plus)
val sumLeft:  List[Int] => Int = _.reverse.foldLeft(0)(flip(plus))

assert( sumRight(list) == sumLeft(list) )
```

```scala
def oneplus[A]: (A,Int) => Int = (x,n) => 1 + n

def lengthRight[A]: List[A] => Int = _.foldRight(0)(oneplus)
def lengthLeft[A]:  List[A] => Int = _.reverse.foldLeft(0)(flip(oneplus))

assert( lengthRight(list) == lengthLeft(list) )
```

```scala
val `(⊕)`: ((Int,Int) => Int) = (n,d) => 10 * n + d

val decimalLeft: List[Int] => Int = _.foldLeft(0)(`(⊕)`)
val decimalRight:  List[Int] => Int = _.reverse.foldRight(0)(flip(`(⊕)`))

assert( decimalLeft(list) == decimalRight(list) )
```

In previous slides we saw a **decimal function** that is implemented with a **right fold**.

It is derived, using the **third duality theorem**, from a **decimal function** implemented with a **left fold**.

```haskell
> n ⊕ d = 10 * n + d
> decimalLeft  = foldl (⊕) 0
> decimalRight = foldr (flip (⊕)) 0 . reverse
```

```scala
val `(⊕)`: ((Int,Int) => Int) = (n,d) => 10 * n + d
val decimalLeft:  List[Int] => Int = _.foldLeft(0)(`(⊕)`)
val decimalRight: List[Int] => Int = _.reverse.foldRight(0)(flip(`(⊕)`))
```

Note how much simpler it is than the **decimal function** that we came up with in **Cheat Sheet #4**.

```haskell
decimal :: [Int] -> Int
decimal ds = fst (foldr f (0,0) ds)

f :: Int -> (Int,Int) -> (Int,Int)
f d (ds, e) = (d * (10 ^ e) + ds, e + 1)
```

```scala
def decimal(ds: List[Int]): Int =
  ds.foldRight((0,0))(f).head

def f(d: Int, acc: (Int,Int)): (Int,Int) = acc match
  case (ds, e) => (d * Math.pow(10, e).toInt + ds, e + 1)
```

That function was produced by the right hand side of the **universal property of** *fold*, after plugging into the left hand side a function that we contrived purely to match that left hand side.

The universal property of *fold*

$$\begin{aligned} g\,[\,] &= v \\ g\,(x : xs) &= f\,x\,(g\,xs) \end{aligned} \quad \Leftrightarrow \quad g = \mathit{fold}\,f\,v$$

$$\begin{aligned} g &:: [\alpha] \to \beta \\ v &:: \beta \\ f &:: \alpha \to \beta \to \beta \end{aligned}$$

Cheat Sheet #6 claimed (see bottom of next slide) that when using **Scala**'s built in **foldRight** function, the reason why doing a **right fold** over a large collection *did not* result in a **stack overflow error**, is that **foldRight** is defined in terms of **foldLeft**.

$$
\begin{aligned}
&\mathit{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\
&\mathit{foldr}\ f\ b\ [\,] = b\\
&\mathit{foldr}\ f\ b\ (x{:}\,xs) = f\ x\ (\mathit{foldr}\ f\ b\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl} :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta\\
&\mathit{foldl}\ f\ b\ [\,] = b\\
&\mathit{foldl}\ f\ b\ (x{:}\,xs) = \mathit{foldl}\ f\ (f\ b\ x)\ xs
\end{aligned}
$$

Scala

```
scala> def foldr[A,B](f: A=>B=>B)(e:B)(s:List[A]):B =
|    s match { case Nil    => e
|              case x::xs => f(x)(foldr(f)(e)(xs)) }

scala> def `(+)`: Long => Long => Long =
     |   m => n => m + n

scala> foldr(`(+)`)(0)(List(1,2,3,4))
val res1: Long = 10

scala> foldr(`(+)`)(0)(List.range(1,10_001))
val res2: Long = 50005000

scala> foldr(`(+)`)(0)(List.range(1,100_001))
java.lang.StackOverflowError

scala> // same again but using built-in function

scala> List.range(1,10_001).foldRight(0)(_+_)
val res3: Int = 50005000

scala> List.range(1,100_001).foldRight(0L)(_+_)
val res4: Long = 5000050000
```

```
scala> import scala.annotation.tailrec

scala> @tailrec
|   def foldl[A,B](f: B=>A=>B)(e:B)(s:List[A]):B =
|     s match { case Nil    => e
|               case x::xs => foldl(f)(f(e)(x))(xs) }

scala> def `(+)`: Long => Long => Long =
     |   m => n => m + n

scala> foldl(`(+)`)(0)(List.range(1,10_001))
val res1: Long = 50005000

scala> foldl(`(+)`)(0)(List.range(1,100_001))
val res2: Long = 5000050000

scala> // same again but using built-in function

scala> List.range(1,10_001).foldLeft(0)(_+_)
val res3: Int = 50005000

scala> List.range(1,100_001).foldLeft(0L)(_+_)
val res4: Long = 5000050000
```

The reason a **stack overflow** is not happening here is because built-in **foldRight** is defined in terms of **foldLeft**! (see cheat-sheet #7)

The remaining slides provide a justification for that claim, and are taken from the following deck, which is what this cheat sheet is based on.

Folding Unfolded

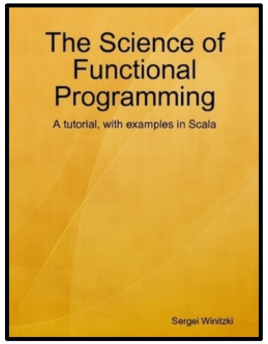
Polyglot FP for Fun and Profit
Haskell and Scala

See **aggregation functions** defined **inductively** and implemented using **recursion**

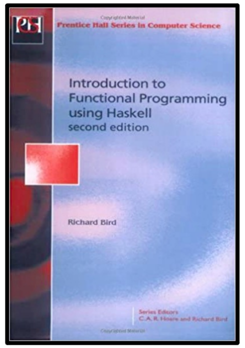
Learn how in many cases, **tail-recursion** and the **accumulator trick** can be used to avoid **stackoverflow errors**

Watch as **general aggregation** is implemented and see **duality theorems** capturing the relationship between **left folds** and **right folds**

Part 2 - through the work of



Sergei Winitzki
sergei-winitzki-11a6431



Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

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

The reason why doing a **right fold** over a large collection *did not* result in a **stack overflow error**, is that the **foldRight** function is implemented by code that **reverses** the **sequence**, **flips** the function that it is passed, and then calls **foldLeft**!

While this is not so obvious when we look at the code for **foldRight** in **List**, because it effectively inlines the call to **foldLeft**…

```scala
final override def foldRight[B](z: B)(op: (A, B) => B): B = {
  var acc = z
  var these: List[A] = reverse
  while (!these.isEmpty) {
    acc = op(these.head, acc)
    these = these.tail
  }
  acc
}
```

```scala
override def foldLeft[B](z: B)(op: (B, A) => B): B = {
  var acc = z
  var these: LinearSeq[A] = coll
  while (!these.isEmpty) {
    acc = op(acc, these.head)
    these = these.tail
  }
  acc
}
```

…it is plain to see in the **foldRight** function for **Seq**

```scala
def foldRight[B](z: B)(op: (A, B) => B): B =
  reversed.foldLeft(z)((b, a) => op(a, b))
```

This is the **third duality theorem** in action

Third duality theorem. For all finite lists xs,

$$foldr\ f\ e\ xs\ =\ foldl\ (flip\ f)\ e\ (reverse\ xs)$$
$$where\ flip\ f\ x\ y = f\ y\ x$$

At the bottom of this slide is where **Functional Programming in Scala** shows that **foldRight** can be defined in terms of **foldLeft**.

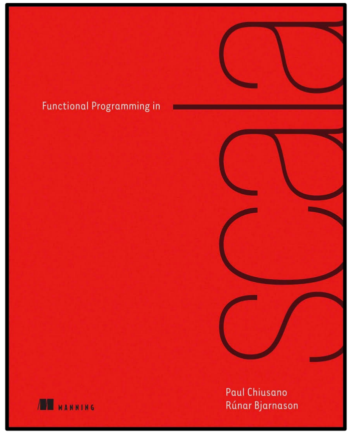
```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```scala
def foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B =
  as match {
    case Nil => z
    case Cons(x, xs) => f(x, foldRight(xs, z)(f))
  }
```

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
1 + (2 + (3 + (foldRight(Nil, 0)((x,y) => x + y))))
1 + (2 + (3 + (0)))
6
```

Our implementation of `foldRight` is not **tail-recursive** and will result in a `StackOverflowError` for large lists (we say it's **not stack-safe**). Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that is **tail-recursive**

```scala
@annotation.tailrec
def foldLeft[A,B](l: List[A], z: B)(f: (B, A) => B): B = l match{
  case Nil => z
  case Cons(h,t) => foldLeft(t, f(z,h))(f) }
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

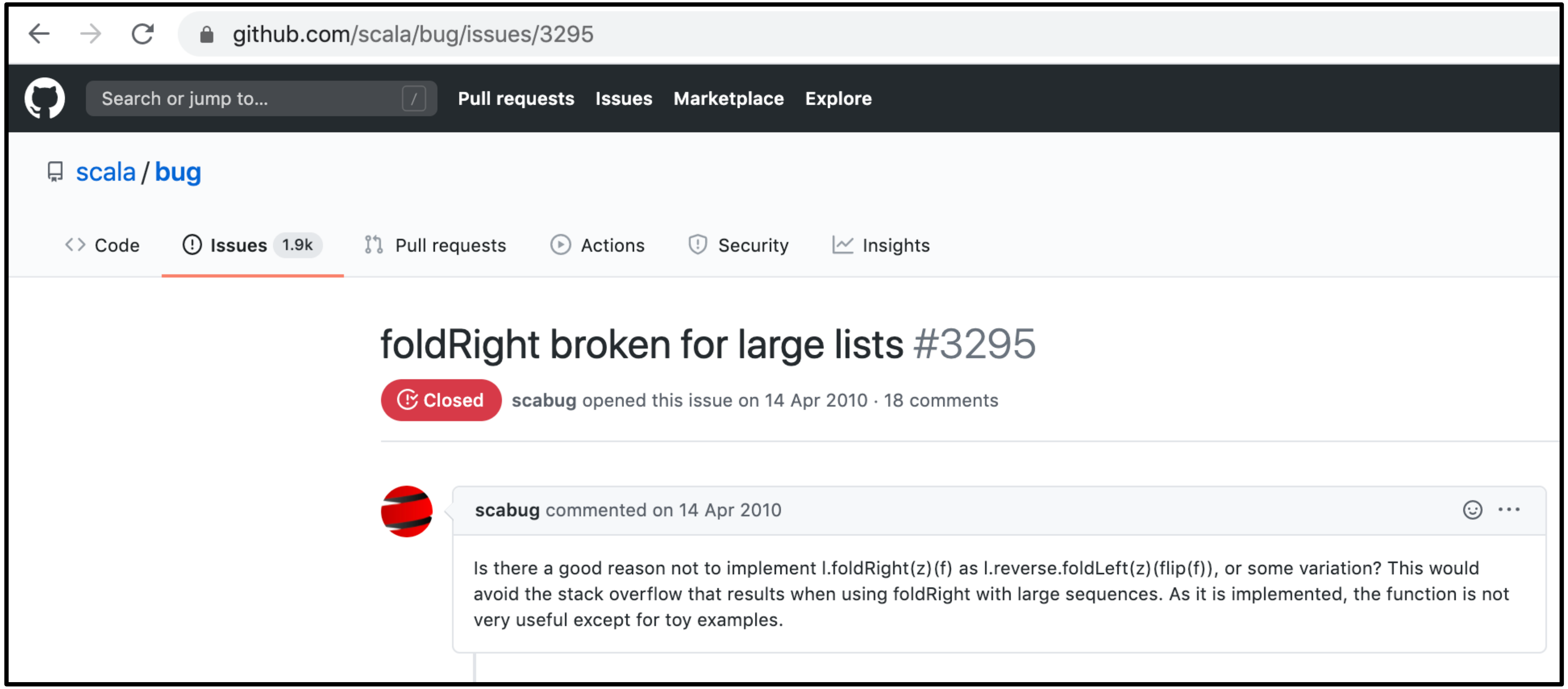
Implementing `foldRight` via `foldLeft` is useful because it lets us implement `foldRight` **tail-recursively**, which means it works even for large lists without overflowing the stack.

```scala
def foldRightViaFoldLeft[A,B](l: List[A], z: B)(f: (A,B) => B): B =
  foldLeft(reverse(l), z)((b,a) => f(a,b))
```

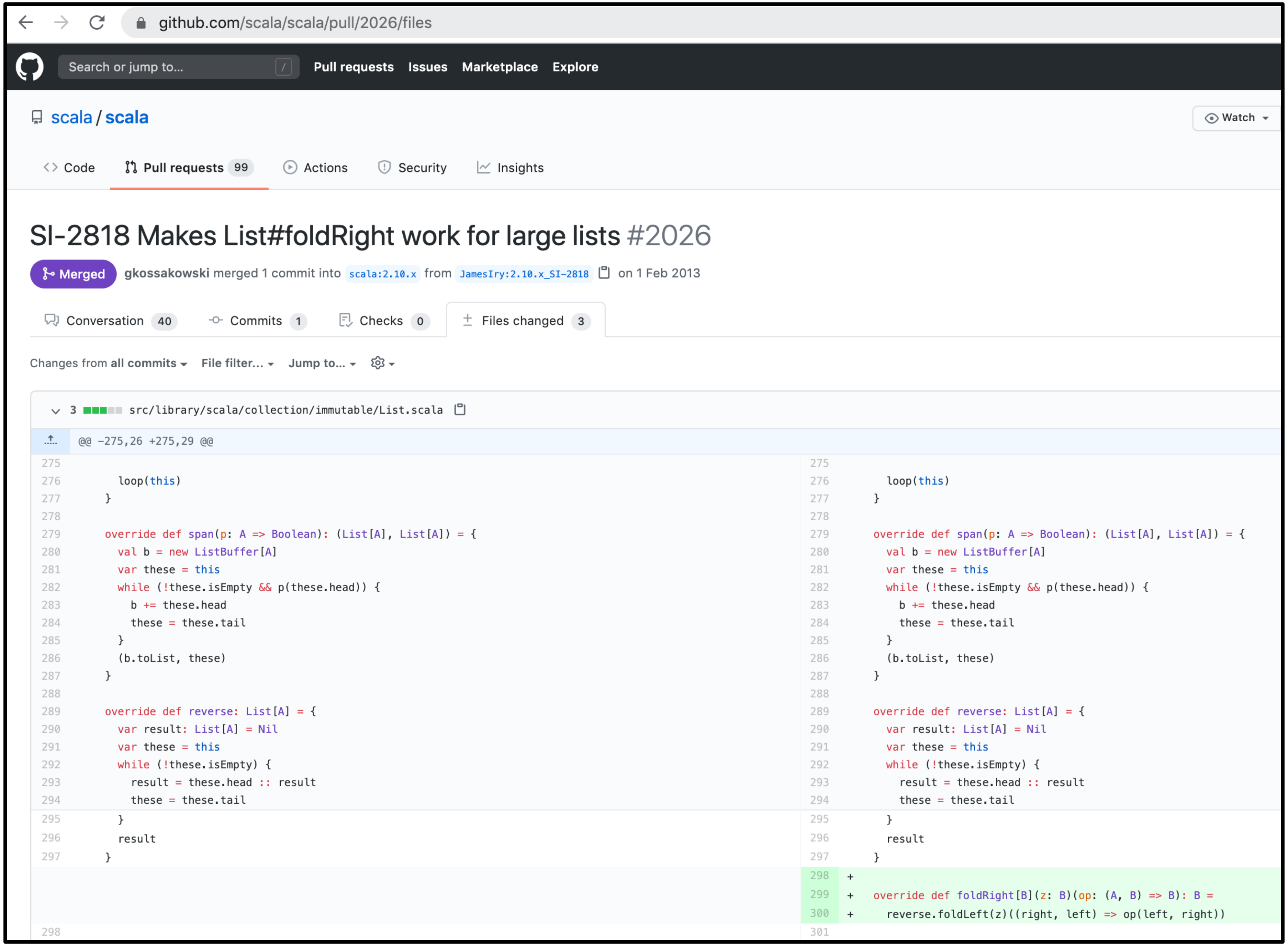
The **third duality theorem** in action.

Scala

It looks like it was none other than **Paul Chiusano** (co-author of FP in Scala), back in 2010, who suggested that `List`'s `foldRight(z)(f)` be implemented as `reverse.foldLeft(z)(flip(f))`!



It also looks like the change was made in 2013 (see next slide) and that it was in 2018 that `foldRight` was reimplemented as a while loop (see slide after that).


github.com/scala/scala/pull/2026/files
Search or jump to...
Pull requests
Issues
Marketplace
Explore
scala / scala
Watch
Code
Pull requests 99
Actions
Security
Insights
SI-2818 Makes List#foldRight work for large lists #2026
Merged
gkossakowski merged 1 commit into scala:2.10.x from JamesIry:2.10.x_SI-2818 on 1 Feb 2013
Conversation 40
Commits 1
Checks 0
Files changed 3
Changes from all commits
File filter...
Jump to...
3 src/library/scala/collection/immutable/List.scala
@@ -275,26 +275,29 @@
    loop(this)
  }

  override def span(p: A => Boolean): (List[A], List[A]) = {
    val b = new ListBuffer[A]
    var these = this
    while (!these.isEmpty && p(these.head)) {
      b += these.head
      these = these.tail
    }
    (b.toList, these)
  }

  override def reverse: List[A] = {
    var result: List[A] = Nil
    var these = this
    while (!these.isEmpty) {
      result = these.head :: result
      these = these.tail
    }
    result
  }
+
+ override def foldRight[B](z: B)(op: (A, B) => B): B =
+   reverse.foldLeft(z)((right, left) => op(left, right))

github.com/scala/scala/commit/878e7d3e0d14633d19bac47dc9b532a54eab6379#diff-65c966843f6b3b817df43968f326d160L486-L487

Search or jump to... / **Pull requests** **Issues** **Marketplace** **Explore**

scala / **scala**

Code Pull requests 99 Actions Security Insights

Migrate collection-strawman into standard library

```
This commit is the result of a scripted migration from the collection-strawman
repository into the main Scala repository. The parent commit is
5b97300 in the master branch of
https://github.com/scala/collection-strawman.git.

The merge commit performs the following changes:
- Move the main strawman sources into the scala.collection namespace under
  src/library/scala/collection. The necessary migration steps have been
  performed and the sources should be fully functional.
- Move the tests to test/collection-strawman. They still need to be integrated
  into the standard test suite in a manual step.
- Delete all other parts (benchmarks, scalafix rules, documentation,
  collections-contrib project) of collection-strawman. They will be moved to
  other repositories.
```

szeiger committed on 22 Mar 2018 — 2 parents 9291e12 + 5b97300 — commit 878e7d3e0d14633d19bac47dc9b532a54eab6379

Showing **371 changed files** with **28,309 additions** and **26,016 deletions**. Unified Split

```
-  override def foldRight[B](z: B)(op: (A, B) => B): B =
-    reverse.foldLeft(z)((right, left) => op(left, right))
-
-  override def stringPrefix = "List"
-
-  override def toStream : Stream[A] =
-    if (isEmpty) Stream.Empty
-    else new Stream.Cons(head, tail.toStream)
```

```
+  final override def foldRight[B](z: B)(op: (A, B) => B): B = {
+    var acc = z
+    var these: List[A] = reverse
+    while (!these.isEmpty) {
+      acc = op(these.head, acc)
+      these = these.tail
+    }
+    acc
+  }
```