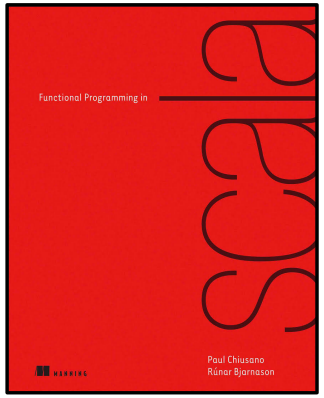
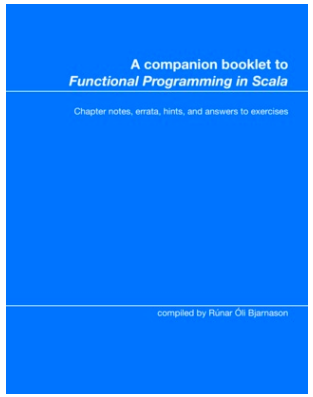
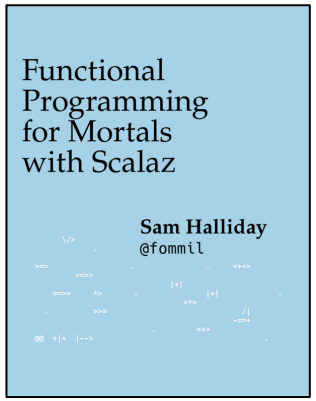
Monoids

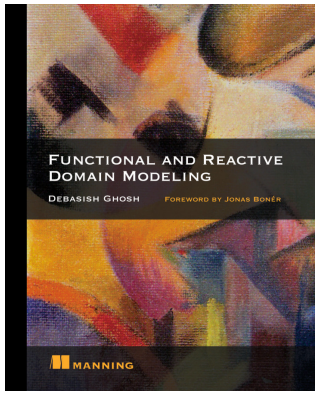
with examples using Scalaz and Cats

Part II - based on









slides by  @philip_schwarz

In **Part 1** I said that while in **Scalaz** there are three **Option monoids**, i.e. **optionFirst**, **optionLast** and **optionMonoid**, in **Cats** there is just one **Option monoid** and it has the same name and behaviour as the **Scalaz optionMonoid**. I would like to correct that before we move on.

Remember how in **Scalaz**, the **op** of the **optionMonoid** combines the content of its **Option** arguments, the **op** of **optionFirst** lets the first non-zero **Option** win, and the **op** of **optionLast** lets the last non-zero **Option** win?

```
import scalaz.Scalaz._

assert(       (2.some |+| 3.some)       == 5.some      ) // optionMonoid combines options by combining their contents
assert( (2.some.first |+| 3.some.first) == 2.some.first) // optionFirst lets the first non-zero option win
assert( (2.some.last  |+| 3.some.last)  == 3.some.last ) // optionLast lets the last non-zero option win
```

@philip_schwarz

And remember how in **Cats** the **op** of **optionMonoid** combines the content of its **Option** arguments, just like in **Scalaz**?

```
import cats.implicits._
import cats.Monoid

val monoid: Monoid[Option[Int]] = cats.kernel.instances.option.catsKernelStdMonoidForOption[Int]
assert( monoid.empty == None)
assert( monoid.combine(Option(2),Option(3)) == Option(5) )
assert( (Option(2) |+| Option(3)) == Option(5) ) // using the |+| alias for the monoid's combine function
```

(Btw, as we see above, what in the **Cats** documentation is called **optionMonoid**, in the **Cats** codebase is called **catsKernelStdMonoidForOption**)

Well, in **Cats** it is also possible to get a **monoid** that behaves like the **Scalaz optionFirst monoid**. Rather than being an instance of **catsKernelStdMonoidForOption**[A] i.e. a **Monoid**[**Option**[A]] with a **combine** function with alias **|+|**, it is **catsStdInstancesForOption**, i.e. a **MonoidK**[**Option**] with a **combineK** function with alias **<+>**:


```
import cats.MonoidK

val monoidK: MonoidK[Option] = cats.instances.option.catsStdInstancesForOption
assert( monoidK.empty == None)
assert( monoidK.combineK(Option(2),Option(3)) == Option(2))
assert( (Option(2) <+> Option(3)) == Option(2) ) // using the <+> alias for the monoidk's combineK function
```

The idea is that while in the first case we have a **Monoid**[A] where A is **Option**[B] and B has a **Semigroup**, in the second case we have a **MonoidK**[F[_]] where F is **Option**. The **K** in **MonoidK** and **combineK** stands for **Kind**, as in Higher-**Kinded** types.

```
scala> :kind -v cats.Monoid
cats.Monoid's kind is F[A]
* -> *
This is a type constructor: a 1st-order-kinded type.

scala> :kind -v cats.MonoidK
cats.MonoidK's kind is X[F[A]]
(* -> *) -> *
This is a type constructor that takes type constructor(s): a higher-kinded type.

scala>
```

While **Monoid** is a type constructor taking a type, e.g. **Option**[**Int**], **MonoidK** is a type constructor that takes a type constructor, e.g. **Option**.

While **Monoid** is a 1st-order-**kinded** type, **MonoidK** is a higher-**kinded** type.

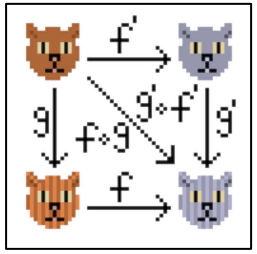
@philip_schwarz

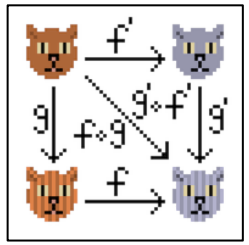
While **catsKernelStdMonoidForOption** is a **Monoid**[**Option**[A]], where A has a **Semigroup**, **catsStdInstancesForOption** is a **MonoidK**[**Option**].

So while the **combine** function of the former knows how to combine the contents of the options it operates on, i.e. by using the **Semigroup**'s **combine** function, the **combineK** function of the latter knows nothing about the type of the contents of the options it operates on and so can only combine the options using their **orElse** function, which results in the first non-zero option winning.

```scala
import simulacrum.typeclass

/**
 * MonoidK is a universal monoid which operates on kinds.
 *
 * This type class is useful when its type parameter F[_] has a
 * structure that can be combined for any particular type, and which
 * also has an "empty" representation. Thus, MonoidK is like a Monoid
 * for kinds (i.e. parametrized types).
 *
 * A MonoidK[F] can produce a Monoid[F[A]] for any type A.
 *
 * Here's how to distinguish Monoid and MonoidK:
 *
 *  - Monoid[A] allows A values to be combined, and also means there
 *    is an "empty" A value that functions as an identity.
 *
 *  - MonoidK[F] allows two F[A] values to be combined, for any A.  It
 *    also means that for any A, there is an "empty" F[A] value. The
 *    combination operation and empty value just depend on the
 *    structure of F, but not on the structure of A.
 */
@typeclass trait MonoidK[F[_]] extends SemigroupK[F] { …
```

```scala
import simulacrum.typeclass

/**
 * SemigroupK is a universal semigroup which operates on kinds.
 *
 * This type class is useful when its type parameter F[_] has a
 * structure that can be combined for any particular type. Thus,
 * SemigroupK is like a Semigroup for kinds (i.e. parametrized
 * types).
 *
 * A SemigroupK[F] can produce a Semigroup[F[A]] for any type A.
 *
 * Here's how to distinguish Semigroup and SemigroupK:
 *
 *  - Semigroup[A] allows two A values to be combined.
 *
 *  - SemigroupK[F] allows two F[A] values to be combined, for any A.
 *    The combination operation just depends on the structure of F,
 *    but not the structure of A.
 */
@typeclass trait SemigroupK[F[_]] { …
```





From https://typelevel.org/cats/typeclasses/semigroupk.html:

There is inline syntax available for both **Semigroup** and **SemigroupK**. Here we are following the convention from scalaz, that **|+|** is the operator from **Semigroup** and that **<+>** is the operator from **SemigroupK** (called **Plus** in **scalaz**).

For **List**, the **Semigroup** instance's **combine** operation and the **SemigroupK** instance's **combineK** operation are both **list concatenation**.

```
assert( (List(1,2) |+| List(3,4) ) == List(1,2,3,4) )
assert( (List(1,2) <+> List(3,4) ) == List(1,2,3,4) )
```

```
assert( (Set("foo","bar") |+| Set("baz")) == Set("foo","bar","baz"))
assert( (Set("foo","bar") <+> Set("baz")) == Set("foo","bar","baz"))
```

i.e. in the case of **List** or **Set**, the behaviour of both **Semigroup**[A].**combine** and **SemigroupK**[F[_]].**combineK** does not rely on the type of the contents of the **List** or **Set**.

However for **Option**, the **Semigroup**'s **combine** and the **SemigroupK**'s **combineK** operation differ. Since **Semigroup** operates on fully specified types, a **Semigroup**[**Option**[A]] knows the concrete type of A and will use **Semigroup**[A].**combine** to combine the inner As. Consequently, **Semigroup**[**Option**[A]].**combine** requires an implicit **Semigroup**[A].

…

In contrast, **SemigroupK**[**Option**] operates on **Option** where the inner type is not fully specified and can be anything (i.e. is "**universally quantified**"). Thus, we cannot know how to combine two of them. Therefore, in the case of **Option** the **SemigroupK**[**Option**].**combineK** method has no choice but to use the **orElse** method of **Option**

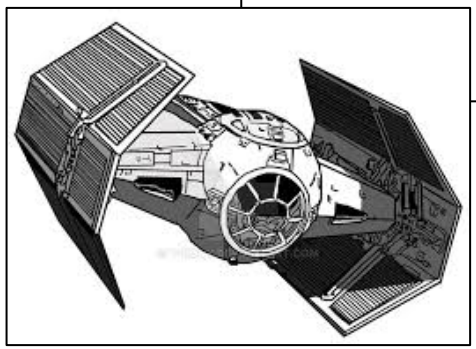
```
val one = Option(1)
val two = Option(2)
val n: Option[Int] = None

assert( (one |+| two) == Some(3))
assert( (one <+> two) == one)

assert( (n |+| two) == two)
assert( (n <+> two) == two)

assert( (two |+| n) == two)
assert( (two <+> n) == two)

assert( (n |+| n) == n)
assert( (n <+> n) == n)
```

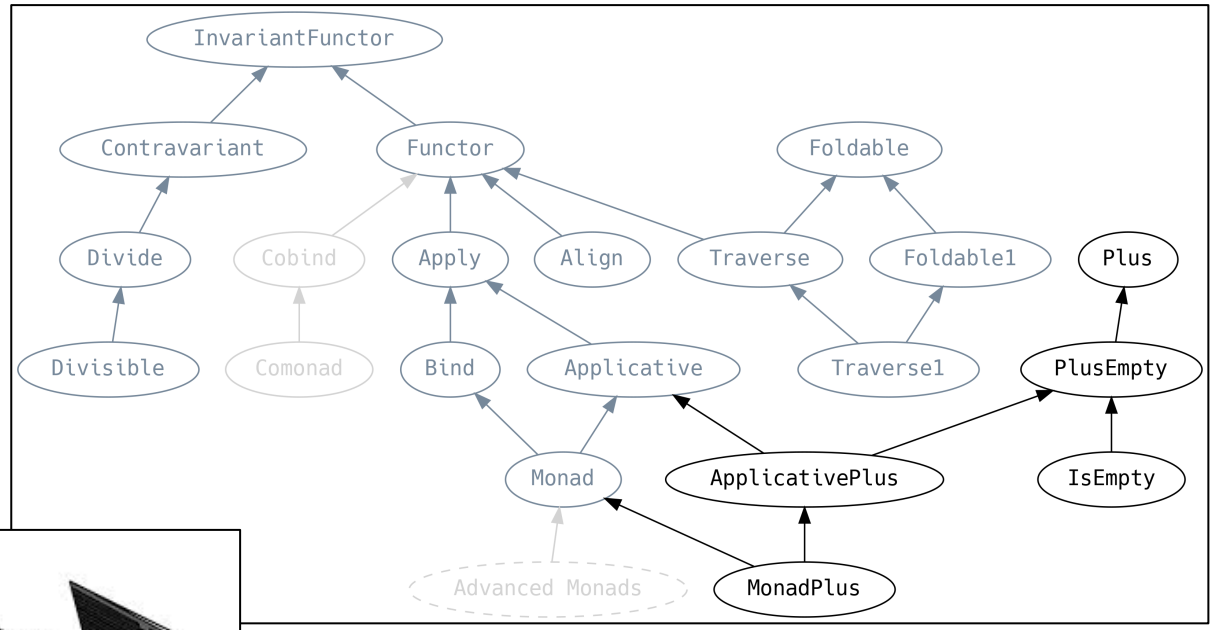
Plus

Plus is **Semigroup** but for type constructors, and **PlusEmpty** is the equivalent of **Monoid** (they even have the same laws) whereas **IsEmpty** is novel and allows us to query if an F[A] is empty:

```
import simulacrum.typeclass
import simulacrum.{op}

@typeclass trait Plus[F[_]] {
  @op("<+>") def plus[A](a: F[A], b: =>F[A]): F[A]
}
@typeclass trait PlusEmpty[F[_]] extends Plus[F] {
  def empty[A]: F[A]
}
@typeclass trait IsEmpty[F[_]] extends PlusEmpty[F] {
  def isEmpty[A](fa: F[A]): Boolean
}
```

<+> is the **TIE Interceptor**, and now we are almost out of **TIE Fighters**.





Although it may look on the surface as if **<+>** behaves like **|+|**:

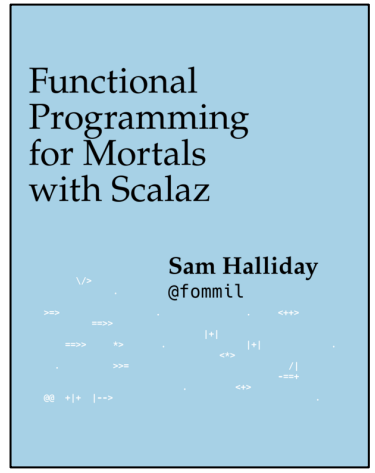
```
scala> List(2,3) |+| List(7)
res0: List[Int] = List(2, 3, 7)

scala> List(2,3) <+> List(7)
res1: List[Int] = List(2, 3, 7)
```

It is best to think of it as operating only at the F[_] level, never looking into the contents.

Plus has the convention that it should **ignore failures** and "**pick the first winner**". **<+>** can therefore be used as a mechanism for **early exit** (losing information) and **failure-handling** via fallbacks:

```
scala> Option(1) |+| Option(2)
res2: Option[Int] = Some(3)

scala> Option(1) <+> Option(2)
res3: Option[Int] = Some(1)

scala> Option.empty[Int] <+> Option(1)
res4: Option[Int] = Some(1)
```

@fommil



Sam Halliday

scalaz.Plus corresponds to **cats.SemigroupK** and **scalaz.PlusEmpty** corresponds to **cats.MonoidK**.

@philip_schwarz

Following that correction regarding **Option monoids** in **Cats**, I would like to look at a **monoid** that is a bit different from the examples we have seen so far, i.e. the **monoid** for **endofunctions**, which is interesting and which we'll anyway need a couple of times later on.

EXERCISE 10.3

A function having the same argument and return type is sometimes called an **endofunction**.[2] Write a **monoid** for **endofunctions**.

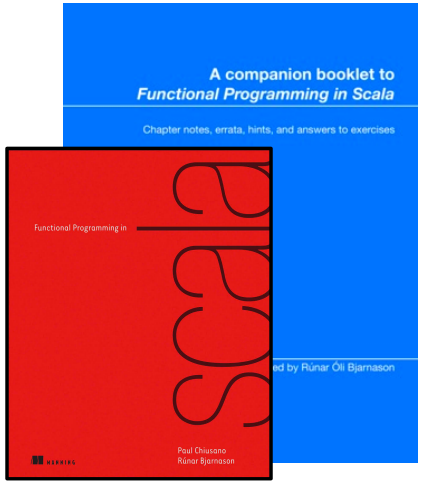
```
def endoMonoid[A]: Monoid[A => A]
```

[2] The Greek prefix **endo-** means **within**, in the sense that an endofunction's **codomain** is within its **domain**.

Again we are limited in the number of ways we can combine values with **op** since it should compose functions of type `A => A` for any choice of `A`. And again there is more than one possible implementation. There is only one possible **zero** though.

There is a choice of implementation here as well. Do we implement it as **f compose g** or **f andThen g**? We have to pick one. We can then get the other one using the **dual** construct.

A Companion booklet to FP in Scala



FP in Scala

```
def endoMonoid[A] = new Monoid[A => A] {
  def op(f: A => A, g: A => A) = f compose g
  val zero = (a: A) => a
}

def dual[A](m: Monoid[A]) = new Monoid[A] {
  def op(x: A, y: A): A = m.op(y, x)
  val zero = m.zero
}
```

```
// the endofunction monoid's zero is the identity function
scala> zero(3)
res0: Int = 3
```

```
// example of monoid laws in action
scala> assert( op(inc, op(twice, square))(3) == op(op(inc, twice), square)(3) )
scala> assert( op(inc, zero)(3) == inc(3) )
scala> assert( op(zero, inc)(3) == inc(3) )
```

```
val intEndoMonoid = endoMonoid[Int]
val intEndoMonoidDual = dual(intEndoMonoid)

val op = intEndoMonoid.op _
val dop = intEndoMonoidDual.op _
val zero = endoMonoid[Int].zero

val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x
val square: Int => Int = x => x * x
```

```
scala> assert( op(inc, twice)(3) == inc(twice(3))) // the monoid's op composes functions
scala> assert( op(inc, twice)(3) == 7)

scala> assert( op(twice, inc)(3) == twice(inc(3))) // try the other way round
scala> assert( op(twice, inc)(3) == 8)

scala> assert( op(zero, op(inc, twice))(3) == 7)   // identity element zero does nothing
scala> assert( op(twice, op(inc, zero))(3) == 8)

scala> assert( op(inc, twice)(3)  == inc(twice(3))) // the op of the monoid is compose
scala> assert( dop(inc, twice)(3) == twice(inc(3))) // the op of the dual monoid is andThen
```

What about in **Scalaz**? Is there a **monoid** for **endofunctions**? Can we compose two **endofunctions** f and g using f |+| g ? Can we expect f |+| **zero** to result in f, if **zero** is the **monoid**'s **identity**?

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

In **Scalaz**, by default, if you have functions `f:A=>B` and `g:A=>B`, then `f|+|g` combines the two functions using a different **monoid** than the **endofunction monoid** we just looked at. From scalaz/example/EndoUsage.scala:

> "there already exists a **Monoid** instance for any `Function1` where there exists a **monoid** for the codomain".

Note: `Function1[A,B]` is just the real object type behind the **syntactic sugar** of function type `A=>B`. If we look at the code in scalaz/std/Function.scala we find out the following:

> **if** for `B`, the codomain of `A=>B`, there exists a **Semigroup** (`B`, **op**)
> **then** for `A=>B`, there exists a **Semigroup** (`A=>B`, **op2**)
> **where op2** takes two functions f: `A=>B` and g: `A=>B` and returns a function `A=>B`
> **such that** given **n**, the function first calls both **f** and **g** with **n** and then combines the results using **op**

e.g.

> **since** for `Int`, the codomain of `Int=>Int`, there exists **Semigroup** (`Int`, **+**)
> **then** for `Int=>Int`, there exists **Semigroup** (`Int=>Int`, **append**)
> **where append** takes two functions `f:Int=>Int` and `g:Int=>Int` and returns a function `Int=>Int`
> **such that** given **n**, the function first calls both **f** and **g** with **n** and then combines the results using **+**

The **semigroup** in question is called **function1Semigroup**. Here is a lambda function capturing the nature of this **semigroup**'s **append** function

```
val append: (Int=>Int, Int=>Int) => (Int=>Int) =
  (f,g) => { n => { f(n) + g(n) } }
```

Using the **function1Semigroup** to combine `Int=>Int` functions **inc** and **twice**

```
val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x
val append = scalaz.std.function.function1Semigroup[Int,Int].append _
assert( append(inc,twice)(3) == inc(3) + twice(3) )
assert( append(inc,twice)(3) == 10 )
```

Since the append function of a **Semigroup** has alias |+|, we can do the following

```
assert( (inc |+| twice)(3) == inc(3) + twice(3) )
assert( (inc |+| twice)(3) == 10 )
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```
val f: Int => Int = inc |+| twice
```

```
val f: Function1[Int,Int] = inc |+| twice
```

```
assert( f(3) == 10 )
```

We saw how the existence of a **Semigroup** (B, **op**) implies the existence of a **Semigroup** (A=>B, **append**)

Similarly, the existence of a **Monoid** (R, **op**, **zero**) implies the existence of a **Monoid** (A=>B, **append**, x => **zero**)
The **zero** of the second **monoid** is a function that always returns the **zero** of the first monoid.

e.g. the existence of **Monoid** (`Int`, **+**, **0**) implies the existence of **Monoid** (`Int=>Int`, **|+|**, (x:`Int`) => **0**)
The **zero** of the second **monoid** is a function that always returns **0**, i.e. the **zero** of the first **monoid**.

The monoid in question is called **function1Monoid**.

```
val zero = scalaz.std.function.function1Monoid[Int,Int].zero
assert( zero(3) == 0 ) // the zero function always returns 0
assert( (inc |+| twice |+| zero)(3) == 10 )
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Another example of using **function1Monoid**

```
val toUpper: String => String = _.toUpperCase
val toLower: String => String = _.toLowerCase

assert( (toUpper |+| toLower)("Scala") == "SCALAscala")

val zero = scalaz.std.function.function1Monoid[String,String].zero

assert( zero("foo") == "") // the zero function always returns ""
assert( (toUpper |+| toLower |+| zero)("Scala") == "SCALAscala")
```

See how the |+| of **Monoid**(String => String, |+|, (x: String)=>"") delegates the task of combining the results of **toUpper** and **toLower** to the |+| of **Monoid**(String,|+|,""), i.e. delegates to it the task of concatenating "SCALA" and "scala".

```
package scalaz
package std

sealed trait FunctionInstances1 {
  implicit def function1Semigroup[A, R](implicit R0: Semigroup[R]): Semigroup[A => R] =
    new Function1Semigroup[A, R] {
      implicit def R = R0
    }
…
}
sealed trait FunctionInstances0 extends FunctionInstances1 {
  implicit def function1Monoid[A, R](implicit R0: Monoid[R]): Monoid[A => R] =
    new Function1Monoid[A, R] {
      implicit def R = R0
    }
…
}
…
private trait Function1Semigroup[A, R] extends Semigroup[A => R] {
  implicit def R: Semigroup[R]

  def append(f1: A => R, f2: => A => R) = a => R.append(f1(a), f2(a))
}

private trait Function1Monoid[A, R] extends Monoid[A => R] with Function1Semigroup[A, R] {
  implicit def R: Monoid[R]
  def zero = a => R.zero
}
…
```

A quick look at where and how **function1Semigroup** and **function1Monoid** are defined

@philip_schwarz

SCALAZ

PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Note how the **Function1Semigroup**'s **append** function **combines** two functions into a function that combines the two functions' results using the **semigroupal append** function.

note how the **monoid**'s **zero** is a function that always returns the **semigroup**'s **zero**.

What about **Cats**? Does it have the equivalent of **function1Semigroup** and **function1Monoid**?
Yes, they are **catsKernelSemigroupForFunction1** and **catsKernelMonoidForFunction1**.

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Cats

```
val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x

import scalaz.std.anyVal.intInstance
import scalaz.std.function.function1Semigroup

val op = function1Semigroup[Int,Int].append _

assert( op(inc,twice)(3) == inc(3) + twice(3) )
assert( op(inc,twice)(3) == 10 )
```

```
import scalaz.syntax.semigroup._ // for |+|

assert( (inc |+| twice)(3) == inc(3) + twice(3) )
assert( (inc |+| twice)(3) == 10 )
```

```
val f: Int => Int =
    inc |+| twice
```

```
val f: Function1[Int,Int] =
    inc |+| twice
```

```
assert( f(3) == 10 )
```

```
import scalaz.std.function.function1Monoid

val zero = function1Monoid[Int,Int].zero
assert( zero(3) == 0 ) // the zero function always returns 0
assert( (inc |+| twice |+| zero)(3) == 10 )
```

```
val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x

import cats.instances.int.catsKernelStdGroupForInt
import cats.instances.function.catsKernelSemigroupForFunction1

val op = catsKernelSemigroupForFunction1[Int,Int].combine _

assert( op(inc,twice)(3) == inc(3) + twice(3) )
assert( op(inc,twice)(3) == 10 )
```

```
import cats.syntax.semigroup._ // for |+|

assert( (inc |+| twice)(3) == inc(3) + twice(3) )
assert( (inc |+| twice)(3) == 10 )
```

```
val f: Int => Int =
    inc |+| twice
```

```
val f: Function1[Int,Int] =
    inc |+| twice
```

```
assert( f(3) == 10 )
```

```
import cats.instances.function.catsKernelMonoidForFunction1

val zero = catsKernelMonoidForFunction1[Int,Int].empty
assert( zero(3) == 0 ) // the zero function always returns 0
assert( (inc |+| twice |+| zero)(3) == 10)
```

So back to this question: in **Scalaz**, is there a **monoid** for **endofunctions**?
Can we compose two **endofunctions** f and g using f |+| g ?
Can we expect f |+| **zero** to result in f if **zero** is the **monoid**'s identity?

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

"The scala **Endo** class is a class which wraps functions from A ⇒ A for some A. This class exists in order to supply some special typeclass instances, since functions where the domain and the codomain are the same type have some special properties."

```scala
/** Endomorphisms.  They have special properties
  * among functions, so are captured in this
  * newtype.
  *
  * @param run The captured function.
  */
final case class Endo[A](run: A => A) {
  final def apply(a: A): A = run(a)

  /** Do `other`, then call myself with its result.*/
  final def compose(other: Endo[A]): Endo[A] =
    Endo.endo(run compose other.run)

  /** Call `other` with my result. */
  final def andThen(other: Endo[A]): Endo[A] =
    other compose this
}

object Endo extends EndoInstances {
…
  /** Alias for `Monoid[Endo[A]].zero`. */
  final def idEndo[A]: Endo[A] = endo[A](a => a)
…
```

```scala
/** Endo forms a monoid where `zero` is the identity endomorphism
  * and `append` composes the underlying functions. */
implicit def endoInstance[A]: Monoid[Endo[A]] =
  new Monoid[Endo[A]] {
    def append(f1: Endo[A], f2: => Endo[A]) = f1 compose f2
    def zero = Endo.idEndo
  }
```

```scala
val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x

assert( (Endo(inc) |+| Endo(twice))(3) == inc(twice(3)))
assert( (Endo(inc) |+| Endo(twice))(3) == 7)

assert( (inc.endo |+| twice.endo)(3) == inc(twice(3)))
assert( (inc.endo |+| twice.endo)(3) == 7)

val f: Endo[Int] = inc.endo |+| twice.endo
assert( f(3) == 7 )
```

```scala
val zero = scalaz.Endo.endoInstance[Int].zero
assert( zero(3) == 3 ) // zero is the identity function
assert( (inc.endo |+| twice.endo |+| zero)(3) == 7 )
```

To select the **dual** of the **endoInstance monoid**, which combines **endofunctions** using **andThen** rather than **compose**, we use the **Dual** tag

```
import scalaz._
import scalaz.Dual._
import scalaz.Scalaz._
import scalaz.Tag.unwrap
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```
scala> val inc: Int => Int = x => x + 1
inc: Int => Int = $$Lambda$3950/177761611@5e763013

scala> inc.endo // select the endomonoid
res0: scalaz.Endo[Int] = Endo($$Lambda$3950/177761611@5e763013)

scala> Dual(inc.endo) // tag the inc endofunction with Dual in order to select the dual of the endomonoid
res1: scalaz.Endo[Int] @@ scalaz.Tags.Dual = Endo($$Lambda$3950/177761611@5e763013)

scala> unwrap(Dual(inc.endo)) // get rid of the Dual tag
res2: scalaz.Endo[Int] = Endo($$Lambda$3950/177761611@5e763013)
```

How to select the **dual** of the **endomonoid** by using the **Dual** tag.

And how to remove the **Dual** tag by using **unwrap**.

```
val inc: Int => Int = x => x + 1
val twice: Int => Int = x => x + x

val incComposeTwice = inc.endo |+| twice.endo // combine functions using compose
assert( incComposeTwice(3) == inc(twice(3)))
assert( incComposeTwice(3) == 7)

val incAndThenTwice = unwrap( Dual(inc.endo) |+| Dual(twice.endo) ) // combine functions using andThen
assert( incAndThenTwice(3) == twice(inc(3)))
assert( incAndThenTwice(3) == 8)
```

First using the **endoInstance monoid** to combine functions using **compose**.

And then using the **dual monoid** to combine functions using **andThen**.

What about **Cats**? Does it have the equivalent of the **scalaz** **endoInstance** **monoid**?

While in **Scalaz**, Endo is a case class with **compose** and **andThen** functions, in **Cats**, Endo is just a type alias: **type** Endo[A] = A => A .

Remember earlier when we used a **MonoidK**[**Option**] to combine two options in first-non-zero-option-wins fashion? In the Scaladoc for **MonoidK** it said that *"A **MonoidK**[F] can produce a **Monoid**[F[A]] for any type A".* Given a **MonoidK**[F] **m**, we can get a **Monoid**[F[Foo]] by calling **m**.**algebra**[Foo].

In **Cats** there is a predefined **MonoidK**[Endo] called **catsStdMonoidKForFunction1**, so one thing we can do is call its **algebra** method for Int to get a **Monoid**[F[Int]], which we can then use to combine **endofunctions** using its **combine** method and using its **|+|** alias.

What we can also do is just use **catsStdMonoidKForFunction1** itself, which being a **MonoidK**[Endo], provides a **combineK** method for combining Endos and a **<+>** alias for this **combineK** function.

```
import cats.Endo

val inc: Endo[Int] = x => x + 1
val twice: Endo[Int] = x => x + x

implicit val endomonoid: cats.Monoid[Endo[Int]] =
  cats.instances.function.catsStdMonoidKForFunction1.algebra[Int]

assert( endomonoid.combine(inc, twice)(3) == 7)

import cats.syntax.semigroup._ // for |+|
assert( (inc |+| twice)(3) == 7)

val zero = endomonoid.empty
assert( (inc |+| twice |+| zero)(3) == 7)
```

```
import cats.Endo

val inc: Endo[Int] = x => x + 1
val twice: Endo[Int] = x => x + x

implicit val endomonoidK: cats.MonoidK[Endo] =
  cats.instances.function.catsStdMonoidKForFunction1

assert( endomonoidK.combineK(inc, twice)(3) == 7)

import cats.syntax.semigroupk._ // for <+>
assert( (inc <+> twice)(3) == 7)

val zero = endomonoidK.empty[Int]
assert( (inc <+> twice <+> zero)(3) == 7)
```

The behaviour of both the following functions depends on the structure of Endo, but not on the structure of Int:

- the **combineK** function of **MonoidK**[Endo]
- the **combine** function of the **Monoid**[F[Int]] created from **MonoidK**[Endo]

If we just want to use the **monoidK**'s **combineK** function through alias **<+>** then we can just do this:

```
import cats.Endo

val inc: Endo[Int] =  x => x + 1
val twice: Endo[Int] = x => x + x

import cats.instances.function._ // for catsStdMonoidKForFunction1
import cats.syntax.semigroupk._  // for <+> of SemigroupK
assert( (inc <+> twice)(3) == 7)

val zero = cats.MonoidK[Endo].empty[Int]
assert( (inc <+> twice <+> zero)(3) == 7)
```

@philip_schwarz

Following that look at **endomonoids**, I would like look at **how to fold lists using monoids**.

So in the following two slides, as background, we look at **FPiS** to recap on **how to fold lists in the first place**.

```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 sum(ints: List[Int]): Int =
  ints match {
    case Nil => 0
    case Cons(x, xs) => x + sum(xs)
}
```

```scala
def product(ds: List[Double]): Double =
  ds match {
    case Nil => 1.0
    case Cons(x, xs) => x * product(xs)
}
```

```
scala> sum(Cons(1,Cons(2,Cons(3,Nil))))
res0: Int = 6
scala> product(Cons(1.0,Cons(2.5,Cons(3.0,Nil))))
res1: Double = 7.5
scala>
```

Note how similar these two definitions are. They're operating on different types (`List[Int]` versus `List[Double]`), but aside from this, **the only differences are the value to return in the case that the list is empty** (`0` in the case of `sum`, `1.0` in the case of `product`), **and the operation to combine results** (`+` in the case of `sum`, `*` in the case of `product`).

Whenever you encounter duplication like this, you can generalize it away by pulling subexpressions out into function arguments…
Let's do that now. Our function will take as arguments the value to return in the case of the empty list, and the function to add an element to the result in the case of a nonempty list.

```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))
  }
```

```scala
def sum(ns: List[Int]) =
  foldRight(ns, 0)((x,y) => x + y)

def product(ns: List[Double]) =
  foldRight(ns, 1.0)(_ * _)
```

`foldRight` is not specific to any one type of element, and we discover while generalizing that the value that's returned doesn't have to be of the same type as the elements of the list!

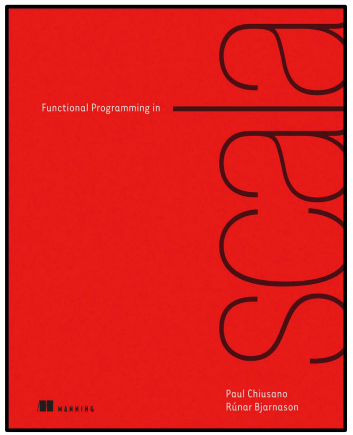
Our implementation of `foldRight` is not tail-recursive and will result in a `StackOverflowError` for large lists (we say it's not stack-safe). Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that is tail-recursive

```
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
```

```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)
}
```

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

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.

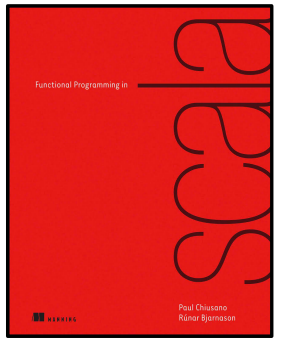


Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

footnotes

[9] In the Scala standard library, `foldRight` is a method on `List` and its arguments are curried similarly for better type inference.

[10] Again, `foldLeft` is defined as a method of `List` in the Scala standard library, and it is curried similarly for better type inference, so you can write `mylist.foldLeft(0.0)(_ + _)`.



FP in Scala

Scala Standard Library 2.12.8 Search

scala.collection.immutable
List

```
def foldLeft[B](z: B)(op: (B, A) ⇒ B): B
```

Applies a binary operator to a start value and all elements of this sequence, going left to right.

Note: will not terminate for infinite-sized collections.

B	the result type of the binary operator.
z	the start value.
op	the binary operator.
returns	the result of inserting op between consecutive elements of this sequence, going left to right with the start value z on the left:

```
op(...op(z, x_1), x_2, ..., x_n)
```

where $x_1, \ldots, x_n$ are the elements of this sequence. Returns z if this sequence is empty.

Definition Classes LinearSeqOptimized → TraversableOnce → GenTraversableOnce

```
def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

Applies a binary operator to all elements of this list and a start value, going right to left.

B	the result type of the binary operator.
z	the start value.
op	the binary operator.
returns	the result of inserting op between consecutive elements of this list, going right to left with the start value z on the right:

```
op(x_1, op(x_2, ... op(x_n, z)...))
```

where $x_1, \ldots, x_n$ are the elements of this list. Returns z if this list is empty.

Definition Classes List → LinearSeqOptimized → IterableLike → TraversableOnce → GenTraversableOnce

```
assert( List(1,2,3,4).foldLeft(0)(_+_) == 10 )
assert( List(1,2,3,4).foldRight(0)(_+_) == 10 )

assert( List(1,2,3,4).foldLeft(1)(_*_) == 24 )
assert( List(1,2,3,4).foldRight(1)(_*_) == 24 )

assert( List("a","b","c","d").foldLeft("")(_+_) == "abcd" )
assert( List("a","b","c","d").foldRight("")(_+_) == "abcd" )
```

After that refresher on **foldLeft** and **foldRight** we can now turn to where **FPiS** explains that **we can fold lists using monoids**.

@philip_schwarz

Folding lists with monoids

Monoids have an intimate connection with lists. If you look at the signatures of **foldLeft** and **foldRight** on **List**, you might notice something about the argument types:

```
def foldRight[B](z: B)(f: (A, B) => B): B
def foldLeft[B](z: B)(f: (B, A) => B): B
```

What happens when A and B are the same type?

```
def foldRight(z: A)(f: (A, A) => A): A
def foldLeft(z: A)(f: (A, A) => A): A
```

The components of a **monoid** fit these argument types like a glove. So if we had a list of **String**s, we could simply pass the **op** and **zero** of the **stringMonoid** in order to reduce the list with the **monoid** and concatenate all the strings:

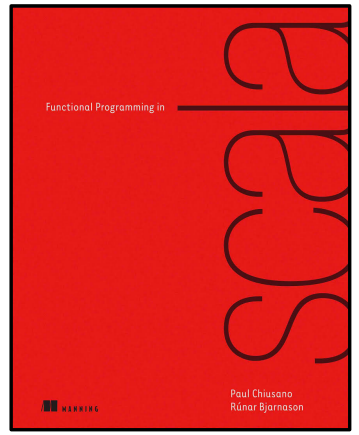
```
scala> val words = List("Hic", "Est", "Index")
words: List[String] = List(Hic, Est, Index)
scala>  val s = words.foldRight(stringMonoid.zero)(stringMonoid.op)
s: String = HicEstIndex
scala> val t = words.foldLeft(stringMonoid.zero)(stringMonoid.op)
t: String = HicEstIndex
scala>
```

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
val stringMonoid: Monoid[String] = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  val zero = ""
}
```

String concatenation function

Note that it doesn't matter if we choose **foldLeft** or **foldRight** when folding with a **monoid**[3]; we should get the same result. This is precisely because the **laws** of **associativity** and **identity** hold. A left fold associates operations to the left, whereas a right fold associates to the right, with the identity element on the left and right respectively:

```
scala> words.foldLeft("")(_ + _) == (("" + "Hic") + "Est") + "Index"
res0: Boolean = true

scala> words.foldRight("")(_ + _) == "Hic" + ("Est" + ("Index" + ""))
res1: Boolean = true
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

[3] Given that both **foldLeft** and **foldRight** have **tail-recursive** implementations.

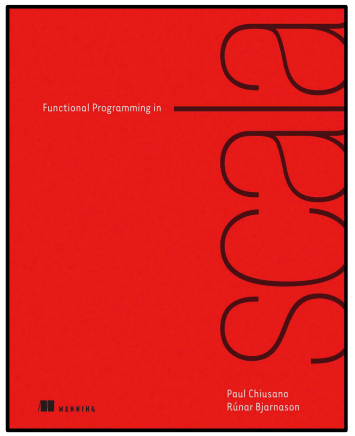
We can write a **general function** `concatenate` that **folds** a **list** with a **monoid**:

```
def concatenate[A](as: List[A], m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.op)
```

But what if our list has an element type that doesn't have a `Monoid` instance? Well, we can always `map` over the list to turn it into a type that does:

```
def foldMap[A, B](as: List[A], m: Monoid[B])(f: A => B): B =
  as.foldLeft(m.zero)((b, a) => m.op(b, f(a)))
```

Notice that this function does not require the use of `map` at all.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

```
val intMonoid = new Monoid[Int] {
  def op(x: Int, y: Int) = x + y
  val zero = 0
}

val stringMonoid = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  val zero = ""
}
```

Let's give **concatenate** and **foldMap** a try using **monoids** for **Int**, **String**, **List**, and **Option**.

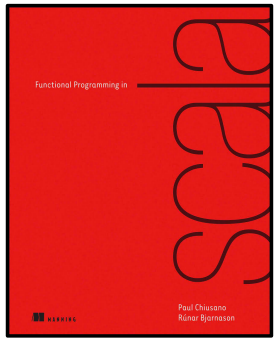
```
def listMonoid[A] = new Monoid[List[A]] {
  def op(a1: List[A], a2: List[A]) = a1 ++ a2
  val zero = Nil
}

def optionMonoid[A] = new Monoid[Option[A]] {
  def op(x: Option[A], y: Option[A]) = x orElse y
  val zero = None
}
```

```
assert( concatenate( List(1,2,3), intMonoid ) == 6 )
assert( concatenate( List("a","b","c"), stringMonoid ) == "abc" )
assert( concatenate( List(List(1,2),List(3,4),List(5,6)), listMonoid[Int]) == List(1,2,3,4,5,6) )
assert( concatenate( List(Some(2), None, Some(3), None, Some(4)), optionMonoid[Int]) == Some(2) )

assert( foldMap( List("1","2","3"), intMonoid )(_ toInt) == 6)
assert( foldMap( List(1, 2, 3), stringMonoid )(_ toString) == "123")
assert( foldMap( List("12","34","56"), listMonoid[Int])(s => (s toList) map (_ - '0')) == List(1,2,3,4,5,6) )
assert( foldMap(List(Some(2), None, Some(3), None, Some(4)), optionMonoid[String])(_ map (_ toString)) == Some("2") )
```

Modifying the definition of **concatenate** to leverage the predefined implicit **monoids** in **Scalaz** and **Cats**.
The same could be done for **foldMap**.

```scala
import scalaz.Monoid
import scalaz.std.anyVal.intInstance
import scalaz.std.string.stringInstance
import scalaz.std.option.optionMonoid
import scalaz.syntax.semigroup._
```

```scala
import cats.Monoid
import cats.instances.int._
import cats.instances.string._
import cats.instances.option._
import cats.syntax.semigroup._
```

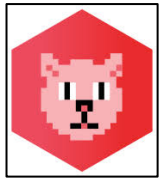


FP in Scala

```scala
def concatenate[A](as: List[A], m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.op)
```

```scala
def concatenate[A: Monoid](as: List[A]): A =
  as.foldLeft(Monoid[A].zero)(_ |+| _)
```

```scala
def concatenate[A: Monoid](as: List[A]): A =
  as.foldLeft(Monoid[A].empty)(_ |+| _)
```



```scala
assert( concatenate(List(2, 3, 4)) == 9)

assert( concatenate(List("2", "3", "4")) == "234")

assert( concatenate(List(Some(2), None, Some(3), None, Some(4))) == Some(9))

assert( concatenate(List(Some("2"), None, Some("3"), None, Some("4"))) == Some("234"))
```

EXERCISE 10.6

Hard: The `foldMap` function can be implemented using either `foldLeft` or `foldRight`. But you can also write `foldLeft` and `foldRight` using `foldMap`! Try it.

Notice that the type of the function that is passed to `foldRight` is `(A,B)=>B`, which can be **curried** to `A=>(B=>B)`. This is a strong hint that we should use the **endofunction monoid** `B=>B` to implement `foldRight`. The implementation of `foldLeft` is then just the **dual**. Don't worry if these implementations are not very efficient.

The function type `(A,B)=>B`, when **curried**, is `A=>(B=>B)`. And of course, `B=>B` is a **monoid** for any `B` (via function composition).

```
def foldRight[A, B](as: List[A])(z:B)(f:(A,B)=>B):B =
  foldMap(as, endoMonoid[B])(f.curried)(z)
```

Folding to the left is the same except we **flip** the arguments to the function `f` to put the `B` on the correct side. Then we have to also "**flip**" the **monoid** so that it operates from left to right.

```
def foldLeft[A, B](as: List[A])(z:B)(f:(B,A)=>B):B =
  foldMap(as, dual(endoMonoid[B]))(a => b => f(b, a))(z)
```

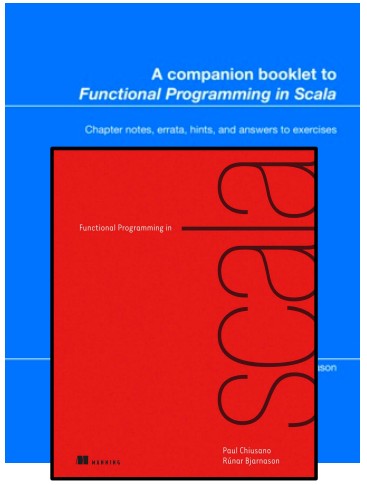
```
def endoMonoid[A] = new Monoid[A=>A]{
  def op(f:A=>A, g:A=>A) = f compose g
  val zero = (a: A) => a
}

def dual[A](m:Monoid[A]) = new Monoid[A]{
  def op(x:A, y:A): A = m.op(y, x)
  val zero = m.zero
}
```

```
def foldMap[A, B](as:List[A],m:Monoid[B])(f:A=>B):B =
  as.foldLeft(m.zero)((b, a) => m.op(b, f(a)))
```

```
def foldMap[A, B](as: List[A], m: Monoid[B])(f: A => B): B =
  as.foldLeft(m.zero)((b, a) => m.op(b, f(a)))

def foldRight[A, B](as: List[A])(z: B)(f: (A, B) => B): B =
  foldMap(as, endoMonoid[B])(f.curried)(z)

def foldLeft[A, B](as: List[A]) (z: B)(f: (B, A) => B): B =
  foldMap(as, dual(endoMonoid[B]))(a => b => f(b, a))(z)

val add: (Int,Int) => Int = _ + _
val multiply: (Int,Int) => Int = _ * _
```

```
assert( foldLeft(List(1,2,3,4))(0)(add) == 10)
assert(foldRight(List(1,2,3,4))(0)(add) == 10)

assert( foldLeft(List(1,2,3,4))(0)(_+_) == 10)
assert(foldRight(List(1,2,3,4))(0)(_+_) == 10)

assert( foldLeft(List(1,2,3,4))(1)(multiply) == 24)
assert(foldRight(List(1,2,3,4))(1)(multiply) == 24)

assert( foldLeft(List(1,2,3,4))(1)(_*_) == 24)
assert(foldRight(List(1,2,3,4))(1)(_*_) == 24)
```

A Companion booklet to FP in Scala



FP in Scala

To better understand the implementation of **foldRight** in terms of **foldMap**, the **endomonoid** and **currying**, let's work through the example of folding a list of integers.

```
def foldRight[A,B](as: List[A])(z:B)(f:(A,B)=>B):B =
  foldMap(as, endoMonoid[B])(f.curried)(z)

def foldMap[A,B](as:List[A],m:Monoid[B])(f:A=>B):B =
  as.foldLeft(m.zero)((b, a) => m.op(b, f(a)))
```

```
scala> foldRight(List(1,2,3))(0)(add)
res10: Int = 6

// unroll, i.e. replace function invocation with function body, substituting formal params with actual params
scala> foldMap(List(1,2,3),endoMonoid[Int])(add.curried)(0)
res11: Int = 6

// unroll
scala> foldLeft(List(1,2,3))(endoMonoid[Int].zero)((b,a)=>(endoMonoid[Int].op(b,add.curried(a))))(0)
res12: Int = 6

// substitute endoMonoid[Int].zero and endoMonoid[Int].op with identity function and compose function
scala> foldLeft(List(1,2,3))(identity[Int] _)((b,a)=>(b compose add.curried(a)))(0)
res13: Int = 6

// unroll
scala> foldLeft(List(2,3))((identity[Int] _) compose add.curried(1))((b,a)=>(b compose add.curried(a)))(0)
res14: Int = 6

// substitute identity[Int] and add.curried(1) with aliases I and add1 (for succinctness)
scala> foldLeft(List(2,3))(I compose add1)((b,a)=>(b compose add.curried(a)))(0)
res15: Int = 6

// unroll
scala> foldLeft(List(3))(I compose add1 compose add2)((b,a)=>(b compose add.curried(a)))(0)
res16: Int = 6

// unroll
scala> foldLeft(Nil)(I compose add1 compose add2 compose add3)((b,a)=>(b compose add.curried(a)))(0)
res17: Int = 6

// unroll
scala> (I compose add1 compose add2 compose add3)(0)
res18: Int = 6
```

```
scala> val add: (Int,Int) => Int = _+_
add: (Int, Int) => Int = …

scala> val I = identity[Int] _
I: Int => Int = …

scala> val add1 = add.curried(1)
add1: Int => Int = …

scala> val add2 = add.curried(2)
add2: Int => Int = …

scala> val add3 = add.curried(3)
add3: Int => Int = …
```

Initially the value of the **foldLeft** accumulator is the identity **endofunction**, but then at each step it gets composed with a new **endofunction** that adds the next list element to its argument. E.g. when the next list element is 2, then **foldLeft** composes accumulator value (**identity** compose **add1**) with **add2**. When the list is empty, **foldLeft** just returns its second argument, i.e. the accumulator.

@philip_schwarz

Great, in that exercise we got a chance to use the **endofunction** monoid that we looked at earlier.

Now let's move on to **Foldable**, an abstraction for things that can be **folded over**, with and without using a **monoid**.

Foldable data structures

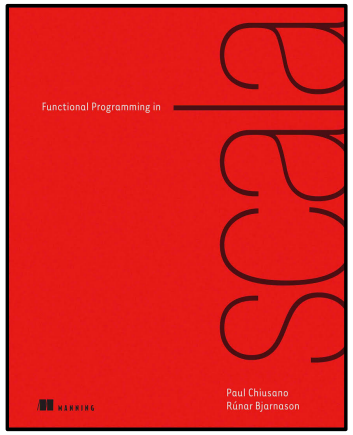
In chapter 3, we implemented the **data structures** `List` and `Tree`, both of which could be **folded**. In chapter 5, we wrote `Stream`, a lazy structure that also can be **folded** much like a `List` can, and now we've just written a **fold** for `IndexedSeq`.

When we're writing code that needs to process data contained in one of these **structures**, **we often don't care about** the **shape** of the **structure** (whether it's a tree or a list), or whether it's **lazy** or not, or provides **efficient random access**, and so forth.

For example, if we have a **structure** full of integers and want to calculate their sum, we can use `foldRight`:

```
ints.foldRight(0)(_ + _)
```

Looking at just this code snippet, **we shouldn't have to care about the type of `ints`**. It could be a `Vector`, a `Stream`, or a `List`, **or anything at all with a `foldRight` method**. **We can capture this commonality in a** `trait`:



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

```
trait Foldable[F[_]] {

  def foldRight[A,B](as: F[A])(z: B)(f: (A,B) => B): B

  def foldLeft[A,B](as: F[A])(z: B)(f: (B,A) => B): B

  def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
      foldLeft(as)(m.zero)(m.op)
}
```

Here we're abstracting over a type constructor `F`, much like we did with the `Parser` type in the previous chapter. We write it as `F[_]`, where the underscore indicates that `F` is not a type but a **type constructor** that takes one type argument. Just like functions that take other functions as arguments are called **higher-order functions**, something like `Foldable` is a **higher-order type constructor** or a **higher-kinded type** .[7]

[7] Just like values and functions have types, types and **type constructors** have **kinds**. Scala uses **kinds** to track how many type arguments a type constructor takes, whether it's co- or contravariant in those arguments, and what the kinds of those arguments are.

EXERCISE 10.12

Implement `Foldable[List]`, `Foldable[IndexedSeq]`, and `Foldable[Stream]`. Remember that **`foldRight`**, **`foldLeft`**, and **`foldMap`** can all be implemented in terms of each other, but that might not be the most efficient implementation.

```
trait Foldable[F[_]] {

  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}

object ListFoldable extends Foldable[List] {
  override def foldRight[A, B](as:List[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:List[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
  override def foldMap[A, B](as:List[A])(f:A=>B)(mb:Monoid[B]):B =
    foldLeft(as)(mb.zero)((b, a) => mb.op(b, f(a)))
}

object IndexedSeqFoldable extends Foldable[IndexedSeq] {…}

object StreamFoldable extends Foldable[Stream] {
  override def foldRight[A, B](as:Stream[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:Stream[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
}
```

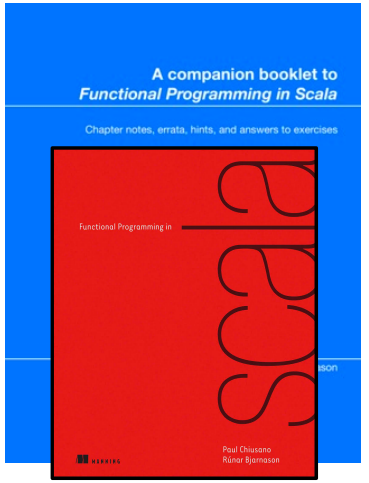
The default implementation of **foldRight** and **foldLeft** use **endoMonoid** and its **dual** respectively.

Using the methods of **ListFoldable** and **StreamFoldable** to fold **Lists**/**Streams** of **Int**s and **String**s.

```
assert( ListFoldable.foldLeft(List(1,2,3))(0)(_+_) == 6)
assert( ListFoldable.foldRight(List(1,2,3))(0)(_+_) == 6)

assert( ListFoldable.concatenate(List(1,2,3))(intMonoid) == 6)
assert( ListFoldable.foldMap(List("1","2","3"))(_ toInt)(intMonoid) == 6)

assert( StreamFoldable.foldLeft(Stream(1,2,3))(0)(_+_) == 6)
assert( StreamFoldable.foldRight(Stream(1,2,3))(0)(_+_) == 6)

assert( StreamFoldable.concatenate(Stream(1,2,3))(intMonoid) == 6)
assert( StreamFoldable.foldMap(Stream("1","2","3"))(_ toInt)(intMonoid) == 6)
```

```
assert( ListFoldable.foldLeft(List("a","b","c"))("")(_+_) == "abc")
assert( ListFoldable.foldRight(List("a","b","c"))("")(_+_) == "abc")

assert( ListFoldable.concatenate(List("a","b","c"))(stringMonoid) == "abc")
assert( ListFoldable.foldMap(List(1,2,3))(_ toString)(stringMonoid) == "123")

assert( StreamFoldable.foldLeft(Stream("a","b","c"))("")(_+_) == "abc")
assert( StreamFoldable.foldRight(Stream("a","b","c"))("")(_+_) == "abc")

assert( StreamFoldable.concatenate(Stream("a","b","c"))(stringMonoid) == "abc")
assert( StreamFoldable.foldMap(Stream(1,2,3))(_ toString)(stringMonoid) == "123")
```

A Companion booklet to FP in Scala



FP in Scala

Next we look at an example of introducing the **Monoid** and **Foldable** abstractions in existing business logic.

@philip_schwarz

4.1.2 Using functional patterns to make domain models parametric

Here's a sample use case from our domain of personal banking that implements backoffice functionality.[4] Clients perform transactions in the form of debits and credits, all of which are logged in the system for auditing and other analytical requirements. You've seen how to manage a client balance as an attribute of an account. Here you'll consider only transactions and balances and **try to implement functionality that allows back-office users to compute various aggregates on transactions executed on a client account**. More specifically, you'll implement the following behaviors in this part of our model:

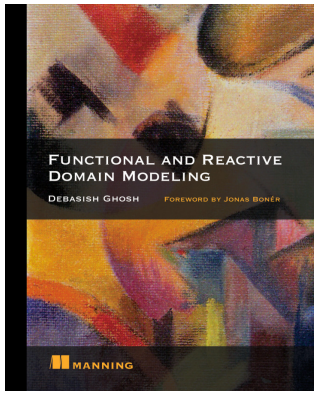
- Given a list of transactions, **you'll identify the highest-value debit transaction that occurred during the day**. Typically, these values may be highlighted as exceptions for auditing purposes.
- Given a list of client balances, **you'll compute the sum of all credit balances**.[5]

All implementations are simple from a domain logic point of view, because the purpose of the implementation is to **identify programming patterns in FP** and not come up with robust, industry-standard models.

IDENTIFYING THE COMMONALITY

So far, in all earlier examples you considered a simple representation of an amount as `BigDecimal`. But in real-life banking, you always need to associate a currency with any amount you specify. So it's time to enrich this part of the model; here we go with our new `Money` model that has both the amount and the currency tagged with it. Not only that, but let's say you ask how much money I have. I check my wallet and say I have 120 U.S. dollars and 25 euros. This means our money abstraction should be able to handle denominations in multiple currencies as well. The following listing contains `Money` and the other base abstractions that you'll use to define the **algebra** of your module.

```
sealed trait TransactionType
case object DR extends TransactionType   debit
case object CR extends TransactionType   credit

sealed trait Currency
case object USD extends Currency
case object JPY extends Currency
case object AUD extends Currency
case object INR extends Currency

object common {
  type Amount = BigDecimal
}

import common._

case class Money(m: Map[Currency, Amount]) {
  def toBaseCurrency: Amount = ???
}

case class Transaction(
  txid: String,
  accountNo: String,
  date: Date,
  amount: Money,
  txnType: TransactionType,
  status: Boolean
)

case class Balance(b: Money)
```



Functional and Reactive Domain Modeling

Debasish Ghosh
@debasishg

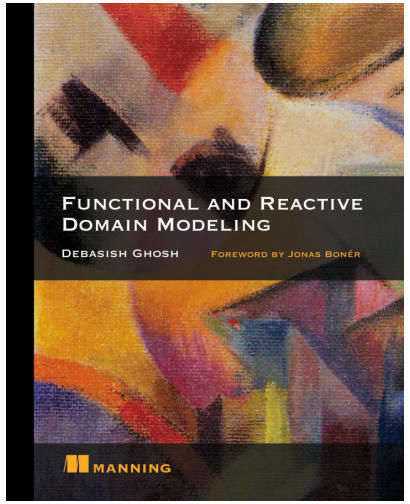
Let's say the behaviors that you'll define belong to a particular module (for example, `Analytics`). Listing 4.2 presents the **algebra** of the module along with a sample **interpreter**.

> **NOTE** Some details of the implementation aren't present in the following listing, but that shouldn't prevent you from understanding its essence. The full runnable source can be found in the online code repository of the book.

```
trait Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction]): Money

  def sumBalances(balances: List[Balance]): Money
}

object Analytics extends Analytics[
  Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction]): Money =
    txns.filter(_.txnType == DR).foldLeft(zeroMoney) {
      (a, txn) =>
        if (gt(txn.amount, a)) valueOf(txn) else a
    }

  def sumBalances(balances: List[Balance]): Money =
    balances.foldLeft(zeroMoney) { (a, b) =>
      a + creditBalance(b)
    }

  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

Functional and Reactive Domain Modeling

https://github.com/debasishg/frdomain



Debasish Ghosh
@debasishg

In the implementation of the `maxDebitOnDay` and `sumBalances` behaviors, **do you see any similarities that you can refactor into more generic patterns**? Let's list some here:

- Both implementations **fold** **over the collection** to compute the core domain logic.
- The **folds take a unit object of `Money` as the seed of the accumulator and perform a binary operation on `Money` as part of the accumulation loop**. In `maxDebitOnDay`, the operation is a **comparison**; in `sumBalances`, it's an **addition**. They are different, but **both are associative and binary**.

I'm sure you see where we're heading—the monoid land. This is the most important part of this exercise: to **look at the pattern and identify the algebra that it fits into**. It won't be a direct fit every time. Sometimes you may have to tweak your implementation to make it fit. But it's all worth it. Instead of implementing a bug-ridden variant of the existing **algebra**, you should always reuse it. These patterns have been refined through the years by experts and field-tested in various production implementations. **The next step is to unify these two seemingly different operations by using the algebra of a monoid**.

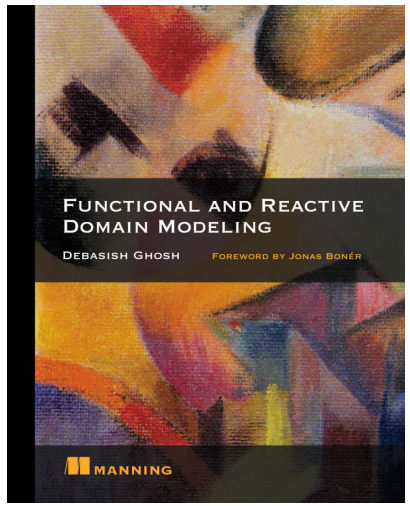
ABSTRACTING OVER THE OPERATIONS

The next step is to define an instance of `Monoid` for `Money`. Because you've defined `Money` in terms of a `Map`, you need to first define `Monoid[Map[K, V]]` and then use that to define `Monoid[Money]`. In fact, you need to define two instances of `Monoid[Money]` because you have two different requirements of operation in `maxDebitOnDay` and `sumBalances`; the former needs an instance based on **comparison** of `Money` and the latter needs one for **addition** of `Money`. Here, for brevity, I'll show the latter one; the one based on comparison is a bit verbose and is implemented in the code base in the online code repository.

```
trait Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money

  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money =
    txns.filter(_.txnType == DR).foldLeft(m.zero) { (a, txn) =>
      m.op(a, valueOf(txn))
    }

  def sumBalances(balances: List[Balance])(implicit m: Monoid[Money]): Money =
    balances.foldLeft(m.zero) { (a, bal) =>
      m.op(a, creditBalance(bal))
    }

  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

```
final val zeroMoney: Money =
  Money(Monoid[Map[Currency, Amount]].zero)

implicit def MoneyAdditionMonoid = new Monoid[Money] {

  val m = implicitly[Monoid[Map[Currency, Amount]]]

  def zero = zeroMoney

  def op(m1: Money, m2: Money) = Money(m.op(m1.m, m2.m))
}
```

Listing4.3 shows the implementation of the `Analytics` module that uses a monoid on `Money`.[6] This is **the first step toward making your model more generic. The operation within the fold is now an operation on a monoid instead of hardcoded operations** on domain-specific types.



Functional and Reactive Domain Modeling

Debasish Ghosh
@debasishg

To better see how **the operation within the fold changed from hardcoded operations on domain-specific types** to **an operation on a monoid**, here is the code before and after the changes, with the modified bits highlighted.

Before

```
trait Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction]): Money

  def sumBalances(bs: List[Balance]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction]): Money =
    txns.filter(_.txnType == DR).foldLeft(zeroMoney) { (a, txn) =>
      if (gt(txn.amount, a)) valueOf(txn) else a
    }

  def sumBalances(balances: List[Balance]): Money =
    balances.foldLeft(zeroMoney) { (a, bal) =>
      a + creditBalance(bal)
    }

  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

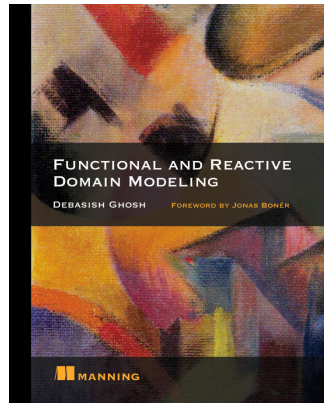
After

```
trait Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money

  def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money
}

object Analytics extends Analytics[Transaction, Balance, Money] {

  def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money =
    txns.filter(_.txnType == DR).foldLeft(m.zero) { (a, txn) =>
      m.op(a, valueOf(txn))
    }

  def sumBalances(balances: List[Balance])(implicit m: Monoid[Money]): Money =
    balances.foldLeft(m.zero) { (a, bal) =>
      m.op(a, creditBalance(bal))
    }

  private def valueOf(txn: Transaction): Money = //..
  private def creditBalance(b: Balance): Money = //..
}
```

ABSTRACTING OVER THE CONTEXT

In the preceding implementation, the first thing that stands out is that in both **`maxDebitOnDay`** and **`sumBalances`**, the action within the **`fold`** is curiously similar. In both cases, **you've abstracted over the operation of the monoid that you passed**. **Because of this abstraction, the code is more generic and needs lesser knowledge of the specific domain elements**.

If you squint hard at both functions, you can see **another similarity**. **In both cases, you `fold` over a collection after mapping through a function that generates a monoid**.




```
def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money =
  txns.filter(_.txnType == DR).foldLeft(m.zero) { (a, txn) =>
    m.op(a, valueOf(txn))
  }
```

`Transaction => Money`

```
def sumBalances(balances: List[Balance])(implicit m: Monoid[Money]): Money =
  balances.foldLeft(m.zero) { (a, bal) =>
    m.op(a, creditBalance(bal))
  }
```

`Balance => Money`

For **`maxDebitOnDay`**, you map using **`valueOf`**, which is `Transaction=>Money`, and for **`sumBalances`** you use **`creditBalance`**, which is `Balance=>Money`.

And `Money` is a **monoid**.[8] If the collection has elements that themselves are **monoids**, you need not do any mapping (or rather you can map with an identity function).

[8] When I say `A` is a **monoid**, I mean that `A` is a type that has a **monoid** instance defined.

In summary, **what you're doing in both functions is, given a collection `F[A]`, which can be folded over, you do a fold on `F[A]`, where either `A` is a monoid or can be mapped into one**. **The only property of the collection that you need is its ability to be folded over**. **So you can make your collection still more generic (and less powerful) by defining it to be a `Foldable[A]`** ; you don't need the richness of a `List[A]` to implement what you need here.

Here's the algebra of your `Foldable` type constructor:

```
trait Foldable[F[_]] {
  def foldl[A, B](as: F[A], z: B, f: (B, A) => B): B
  def foldMap[A, B](as: F[A])(f: A => B)(implicit m: Monoid[B]): B =
    foldl(as, m.zero, (b: B, a: A) => m.op(b, f(a)))
}
```

Debasish Ghosh
@debasishg

```scala
trait Foldable[F[_]] {
  def foldl[A,B](as: F[A], z: B, f: (B, A) => B): B
  def foldMap[A,B](as:F[A])(f:A => B)(implicit m:Monoid[B]): B =
    foldl(as, m.zero, (b: B, a: A) => m.op(b, f(a)))
}
```

```scala
implicit val listFoldable = new Foldable[List] {
  def foldl[A,B](as: List[A], z:B, f: (B,A) => B) =
    as.foldLeft(z)(f)
}
```

The function **foldMap** does exactly what I said before: **folds** over a collection `F[A]`, where `f: A=> B` generates a **monoid** `B` out of `A`. And **f** can be an **identity** if `A` is a **monoid**. So, given a `Foldable[A]`, a type `B` that's a **monoid**, and a mapping function between `A` and `B`, **you can package foldMap nicely into a combinator that abstracts your requirements of maxDebitOnDay and sumBalances (and many other similar domain behaviors) without sacrificing the holy grail of parametricity**. And **this is the second step toward making your model more generic using design patterns: You've abstracted over the context, the type constructor of your abstraction**.

```scala
  def mapReduce[F[_],A,B](as: F[A])(f: A => B)(implicit fd: Foldable[F], m: Monoid[B]) =
    fd.foldMap(as)(f)
```

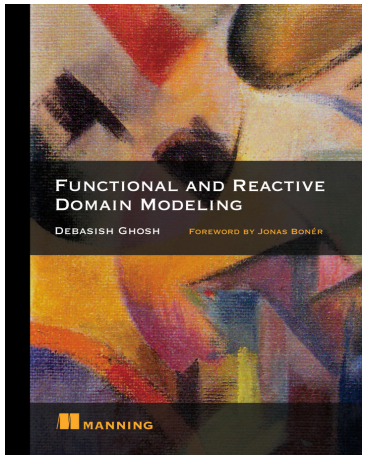
And now **each of your module functions becomes as trivial as a one-liner**:

```scala
  object Analytics extends Analytics[Transaction, Balance, Money] {

    def maxDebitOnDay(txns: List[Transaction])(implicit m: Monoid[Money]): Money =
      mapReduce(txns.filter(_.txnType == DR))(valueOf)(implicit foldable)

    def sumBalances(bs: List[Balance])(implicit m: Monoid[Money]): Money =
      mapReduce(bs)(creditBalance)(implicit foldable)
  }
```

`Transaction => Money`

`Balance => Money`

The complete runnable code of this entire exercise can be found in the online code repository for the book.



Debasish Ghosh
@debasishg

What is the marketing buzzword for **foldMap**?
See the next slide.

@philip_schwarz

5.4.2 Foldable

Technically, **Foldable is for data structures that can be walked to produce a summary value**. However, **this undersells the fact that it is a one-typeclass army that can provide most of what you'd expect to see in a Collections API**.

There are so many methods we are going to have to split them out, beginning with the abstract methods:

```
@typeclass trait Foldable[F[_]] {
  def foldMap[A, B: Monoid](fa: F[A])(f: A => B): B
  def foldRight[A, B](fa: F[A], z: =>B)(f: (A, =>B) => B): B
  def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B = ...
```

An instance of **Foldable** need only implement **foldMap** and **foldRight** to get all of the functionality in this typeclass, although methods are typically optimised for specific data structures.

You might recognise foldMap by its marketing buzzword name, MapReduce. Given an F[A], a function from A to B, and a way to combine B (provided by the **Monoid**, along with a **zero** B), we can produce a summary value of type B. **There is no enforced operation order, allowing for parallel computation**.

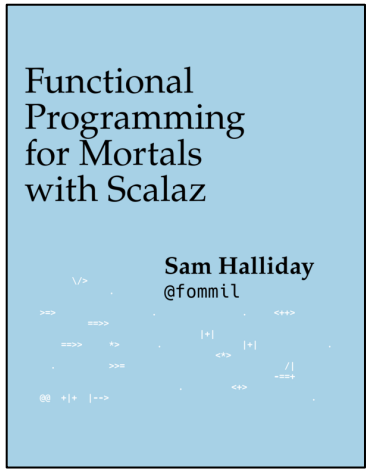
foldRight does not require its parameters to have a **Monoid**, meaning that it needs a starting value **z** and a way to **combine** each element of the data structure with the summary value. The order for traversing the elements is from right to left and therefore it cannot be parallelised.

foldLeft traverses elements from left to right. **foldLeft** can be implemented in terms of **foldMap**, but most instances choose to implement it because it is such a basic operation. Since it is usually implemented with tail recursion, there are no byname parameters.

The only law for Foldable is that foldLeft and foldRight should each be consistent with foldMap for monoidal operations. e.g. appending an element to a list for **foldLeft** and prepending an element to a list for **foldRight**. However, **foldLeft** and **foldRight** do not need to be consistent with each other: in fact they often produce the reverse of each other.

The simplest thing to do with **foldMap** is to use the identity function, giving **fold** (the natural sum of the monoidal elements), with left/right variants to allow choosing based on performance criteria:

```
  def fold[A: Monoid](t: F[A]): A = ...
  def sumr[A: Monoid](fa: F[A]): A = ...
  def suml[A: Monoid](fa: F[A]): A = ...
...
```



Sam Halliday

You might rtecognize **foldMap** by its marketing name, **MapReduce**.

@fommil

The four fundamental functions of the the Foldable trait in FPiS, Scalaz and Cats

fold

FPiS
```
def concatenate[A](as: F[A])(m: Monoid[A]): A =
  foldLeft(as)(m.zero)(m.op)
```

Scalaz
```
def fold[M:Monoid](t:F[M]):M =
  foldMap[M, M](t)(x => x)
```
```
def sumr[A](fa:F[A])(implicit A:Monoid[A]):A =
  foldRight(fa, A.zero)(A.append)
```
```
def suml[A](fa:F[A])(implicit A: Monoid[A]): A =
  foldLeft(fa, A.zero)(A.append(_, _))
```

Cats
```
def fold[A](fa: F[A])(implicit A: Monoid[A]): A =
  foldLeft(fa, A.empty) { (acc, a) => A.combine(acc, a) }
```
```
def combineAll[A: Monoid](fa: F[A]): A =
  fold(fa)
```

foldMap

FPiS
```
def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
  foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))
```

Scalaz
```
def foldMap[A,B](fa: F[A])(f: A => B)(implicit F: Monoid[B]): B
```

Cats
```
def foldMap[A, B](fa: F[A])(f: A => B)(implicit B: Monoid[B]): B =
  foldLeft(fa, B.empty)((b, a) => B.combine(b, f(a)))
```

foldRight

FPiS
```
def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
  foldMap(as)(f.curried)(endoMonoid[B])(z)
```

Scalaz
```
def foldRight[A, B](fa: F[A], z: => B)(f: (A, => B) => B): B
```

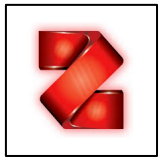
Cats
```
def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```

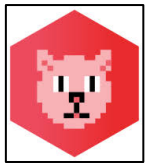
foldLeft

FPiS
```
def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
  foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)
```

Scalaz
```
def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B = {
  import Dual._, Endo._, syntax.std.all._
  Tag.unwrap(foldMap(fa)((a: A) =>
    Dual(Endo.endo(f.flip.curried(a))))(dualMonoid)) apply (z)
}
```

Cats
```
def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B
```





concatenate	fold,suml,sumr	fold,combineAll
foldMap	foldMap	foldMap
foldLeft	foldLeft	foldLeft
foldRight	foldRight	foldRight

Let's take the **fold** and **foldMap** of **Cats'** **Foldable** for a spin.

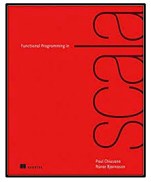
It is simpler to start off by using **combineAll** rather than **fold** because the latter clashes with the **fold** in the Scala standard library.

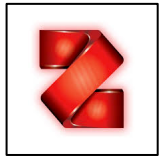
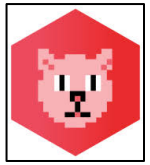
```
import cats.Monoid
import cats.Foldable
import cats.instances.int._
import cats.instances.string._
import cats.instances.option._
import cats.instances.list._
import cats.syntax.foldable._

assert( List(1,2,3).combineAll == 6 )
assert( List("a","b","c").combineAll == "abc" )
assert( List(List(1,2),List(3,4),List(5,6)).combineAll == List(1,2,3,4,5,6) )
assert( List(Some(2), None, Some(3), None, Some(4)).combineAll == Some(9) )

assert( List("1","2","3").foldMap(_ toInt) == 6)
assert( List(1, 2, 3).foldMap(_ toString) == "123")
assert( List("12","34","56").foldMap( s => (s toList) map (_ - '0')) == List(1,2,3,4,5,6) )
assert( List(Some(2), None, Some(3), None, Some(4)).foldMap(_ toList) == List(2,3,4) )

// when we call fold on a List we call the fold in the Scala Standard library
List(1,2,3).fold(0)(_ + _)

// but when we call fold on a Foldable we call the Cats fold
def businessLogic[A:Monoid,F[_]: Foldable](foldable:F[A]): A =
  /*...*/ foldable.fold /*...*/

def assertFoldEquals[A:Monoid,F[_]: Foldable](foldable:F[A], expectedValue:A) =
  assert(foldable.fold == expectedValue)

assertFoldEquals(List(1,2,3), 6)
assertFoldEquals(List("a","b","c"), "abc")
assertFoldEquals(List(List(1,2),List(3,4),List(5,6)), List(1,2,3,4,5,6))
assertFoldEquals(List(Some(2), None, Some(3), None, Some(4)), Some(9))
```

And here we do the same thing using **Scalaz**. The only differences with **Cats** are marked in yellow.

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```
import scalaz.Monoid
import scalaz.Foldable
import scalaz.std.anyVal.intInstance
import scalaz.std.string.stringInstance
import scalaz.std.option.optionMonoid
import scalaz.std.list.listInstance
import scalaz.std.list.listMonoid
import scalaz.syntax.foldable.ToFoldableOps

assert( List(1,2,3).concatenate == 6 )
assert( List("a","b","c").concatenate == "abc" )
assert( List(List(1,2),List(3,4),List(5,6)).concatenate == List(1,2,3,4,5,6) )
assert( List(Some(2), None, Some(3), None, Some(4)).concatenate == Some(9) )

assert( List("1","2","3").foldMap(_ toInt) == 6)
assert( List(1, 2, 3).foldMap(_ toString) == "123")
assert( List("12","34","56").foldMap( s => (s toList) map (_ - '0')) == List(1,2,3,4,5,6) )
assert( List(Some(2), None, Some(3), None, Some(4)).foldMap(_ toList) == List(2,3,4) )

// when we call fold on a List we call the fold in the Scala Standard library
List(1,2,3).fold(0)(_ + _)

// but when we call fold on a Foldable we call the Scalaz fold
def businessLogic[A:Monoid,F[_]: Foldable](foldable:F[A]): A =
  /*...*/ foldable.fold /*...*/

def assertFoldEquals[A:Monoid,F[_]: Foldable](foldable:F[A], expectedValue:A) =
  assert(foldable.fold == expectedValue)

assertFoldEquals(List(1,2,3), 6)
assertFoldEquals(List("a","b","c"), "abc")
assertFoldEquals(List(List(1,2),List(3,4),List(5,6)), List(1,2,3,4,5,6))
assertFoldEquals(List(Some(2), None, Some(3), None, Some(4)), Some(9))
```



concatenate	fold,suml,sumr	fold,combineAll
foldMap	foldMap	foldMap
foldLeft	foldLeft	foldLeft
foldRight	foldRight	foldRight

+

Note that here we are using **concatenate**, which is a **fold** alias defined in **FoldableOps**. This is similar to **Cats** providing **fold** alias **combineAll**, except that in that case the alias is defined in **Foldable** itself.

to be continued in part 3