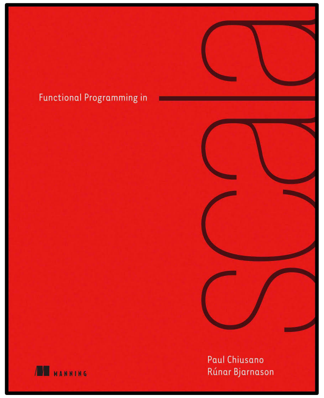
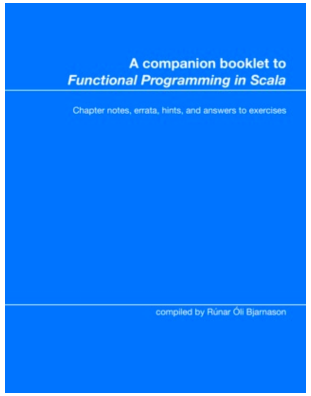
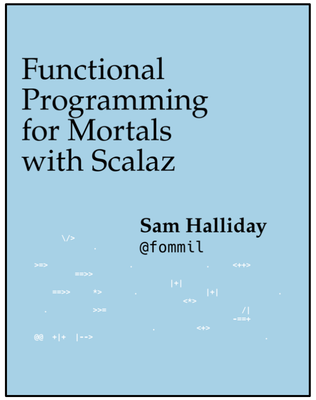
Monoids

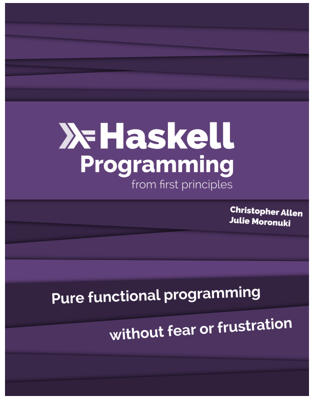
with examples using Scalaz and Cats

based on









Part 1

slides by  @philip_schwarz

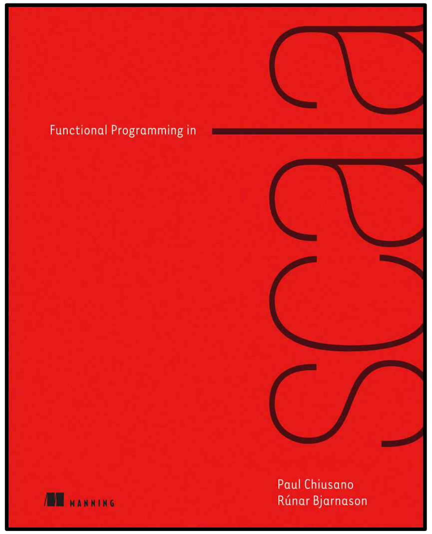
What is a monoid?

Let's consider the algebra of **string concatenation**. We can add `"foo" + "bar"` to get `"foobar"`, and the **empty string** is an **identity element** for that operation. That is, if we say `(s + "")` or `("" + s)`, the result is always `s`.

```
scala> val s = "foo" + "bar"
s: String = foobar

scala> assert( s == s + "" )

scala> assert( s == "" + s )

scala>
```

Furthermore, if we combine three strings by saying `(r + s + t)`, the operation is **associative** —it doesn't matter whether we parenthesize it: `((r + s) + t)` or `(r + (s + t))`.

```
scala> val (r,s,t) = ("foo","bar","baz")
r: String = foo
s: String = bar
t: String = baz

scala> assert( ( ( r + s ) + t ) == ( r + ( s + t ) ) )

scala> assert( ( ( r + s ) + t ) == "foobarbaz" )

scala>
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

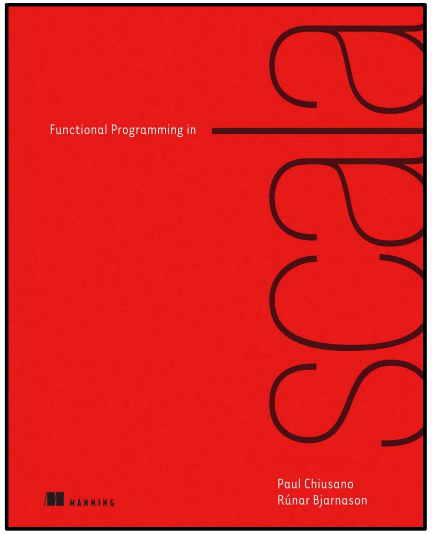
@pchiusano @runarorama

The exact same rules govern **integer addition**. It's **associative**, since `(x + y) + z` is always equal to `x + (y + z)`

```
scala> val (x,y,z) = (1,2,3)
x: Int = 1
y: Int = 2
z: Int = 3

scala> assert( ( ( x + y ) + z ) == ( x + ( y + z ) ) )

scala> assert( ( ( x + y ) + z ) == 6 )

scala>
```

and it has an **identity element**, `0` , which "does nothing" when added to another integer

```
scala> val s = 3
s: Int = 3

scala> assert( s == s + 0)

scala> assert( s == 0 + s)

scala>
```



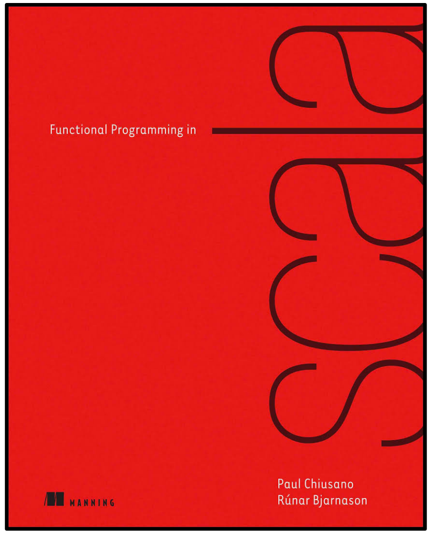
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Ditto for **integer multiplication**

```
scala> val (x,y,z) = (2,3,4)
x: Int = 2
y: Int = 3
z: Int = 4

scala> assert( ( ( x * y ) * z ) == ( x * ( y * z ) ) )

scala> assert( ( ( x * y ) * z ) == 24 )

scala>
```

whose **identity element** is **1**

```
scala> val s = 3
s: Int = 3

scala> assert( s == s * 1)

scala> assert( s == 1 * s)

scala>
```



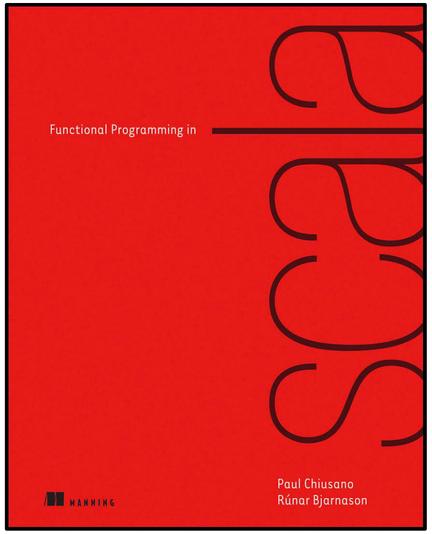
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

The **Boolean** operators `&&` and `||` are likewise **associative**

```
scala> val (p,q,r) = (true,false,true)
p: Boolean = true
q: Boolean = false
r: Boolean = true

scala> assert( ( ( p || q ) || r ) == ( p || ( q || r ) ) )

scala> assert( ( ( p || q ) || r ) == true )

scala> assert( ( ( p && q ) && r ) == ( p && ( q && r ) ) )

scala> assert( ( ( p && q ) && r ) == false )

scala>
```

and they have **identity elements** `true` and `false`, respectively

```
scala> val s = true
s: Boolean = true

scala> assert( s == ( s && true ) )

scala> assert( s == ( true && s ) )

scala> assert( s == ( s || false ) )

scala> assert( s == ( false || s ) )

scala>
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

These are just a few simple examples, but algebras **like this are virtually** **<u>everywhere</u>**. The term for this kind of algebra is **monoid**.

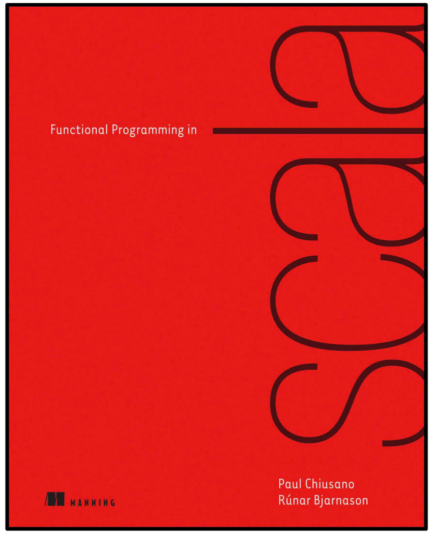
The **laws** of **associativity** and **identity** are collectively called the **monoid laws**.

A **monoid** consists of the following:

- Some type `A`
- An associative binary operation, `op`, that takes two values of type `A` and combines them into one: `op(op(x,y), z) == op(x, op(y,z))` for any choice of `x: A`, `y: A`, `z: A`
- A value, `zero: A`, that is an identity for that operation: `op(x, zero) == x` and `op(zero, x) == x` for any `x: A`

We can express this with a Scala trait:

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

An example instance of this trait is the `String` **monoid**:

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

String concatenation function

`List` concatenation also forms a **monoid**:

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

List function returning a new list containing the elements from the left hand operand followed by the elements from the right hand operand

Monoid instances for **integer addition** and **multiplication** as well as the **Boolean operators**

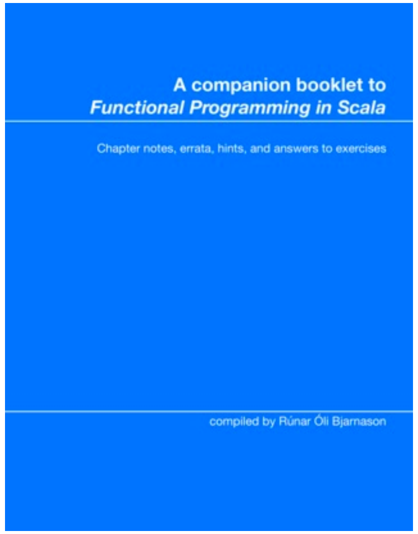
```scala
val intAddition: Monoid[Int] = new Monoid[Int] {
    def op(x: Int, y: Int) = x + y
    val zero = 0
}

val intMultiplication: Monoid[Int] = new Monoid[Int] {
    def op(x: Int, y: Int) = x * y
    val zero = 1
}
```

```scala
val booleanOr: Monoid[Boolean] = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x || y
    val zero = false
}

val booleanAnd: Monoid[Boolean] = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x && y
    val zero = true
}
```

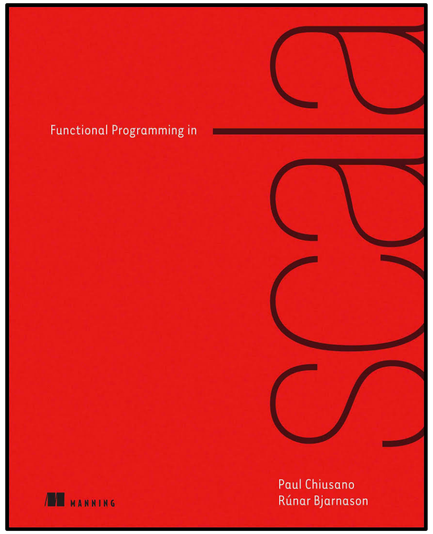
Just what is a **monoid**, then? It's simply a type `A` and **an implementation of `Monoid[A]`** that satisfies the **laws**.

Stated tersely, a **monoid** is a **type** together with a **binary operation** (**op**) over that type, satisfying **associativity** and having an **identity element** (**zero**).

What does this buy us? **Just like any abstraction, a monoid is useful to the extent that we can write useful generic code assuming only the capabilities provided by the abstraction. Can we write any interesting programs, knowing nothing about a type other than that it forms a monoid? Absolutely**!



(by Runar Bjarnason)
@runarorama



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

@philip_schwarz

Here is a very simple, contrived example of a **generic function** called **combine** that operates on any three values of a type A for which an **implicit monoid** is available.

It takes each of three pairs of values and produces a **combined** value for the pair by applying the **monoid**'s **binary operation** to the pair's elements, returning a tuple of the resulting **combined** values.

```
def combine[A](a: A, b: A, c: A)(implicit m: Monoid[A]): (A,A,A) =
  ( m.op(a,b), m.op(a,c), m.op(b,c) )
```

If we now revisit some of the **monoid instances** we defined earlier and declare them to be **implicit**, we can then invoke our generic **combine** function multiple times, each time passing in values of a different type, and each time implicitly passing in a **monoid instance** associated with that type.

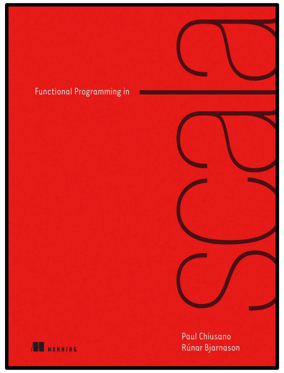
```
implicit val stringMonoid = new Monoid[String] …
implicit def listMonoid[A] = new Monoid[List[A]] …
implicit val intAddition  = new Monoid[Int] …
```

```
scala>  assert( combine("a","b","c") == ("ab","ac","bc"))
scala>  assert( combine(List(1,2),List(3,4),List(5,6)) == (List(1,2,3,4),List(1,2,5,6),List(3,4,5,6)) )
scala>  assert( combine(1,2,3) == (3,4,5) )
scala>
```

What about **Scalaz**? **Scalaz** provides a predefined **Monoid** trait whose **binary operation** is called **append**, rather than **op**, and provides **predefined implicit instances**, e.g. for String, List and integer addition. So all we have to do is add a couple of imports and we can then define **combine** as follows:

```
import scalaz.Scalaz._
import scalaz._

def combine[A](a: A, b: A, c: A)(implicit m: Monoid[A]): (A,A,A) =
  ( m.append(a,b), m.append(a,c), m.append(b,c) )
```

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



FP in Scala

In Scalaz the binary operation is called append, rather than op and it is not defined in the Monoid trait, but in the Semigroup trait, which the Monoid trait extends.

@philip_schwarz

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```scala
trait Semigroup[F]  { self =>
  def append(f1: F, f2: => F): F
  …
```

```scala
trait Monoid[F] extends Semigroup[F] { self =>
  def zero: F
  …
```

```scala
final class SemigroupOps[F]…(implicit val F: Semigroup[F]) … {
  final def |+|(other: => F): F      = F.append(self, other)
  final def mappend(other: => F): F = F.append(self, other)
  final def ⊹(other: => F): F        = F.append(self, other)
…
```

and the SemigroupOps class defines three aliases of append that are infix operators: |+|, mappend, ⊹

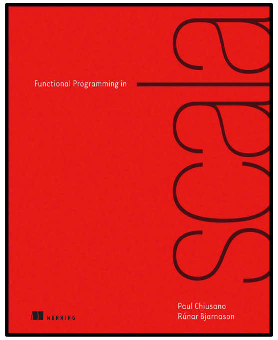
So our combine function can just take an implicit Semigroup rather than an implicit Monoid

```scala
def combine[A](a: A, b: A, c: A)(implicit sg: Semigroup[A]): (A,A,A) = ???
```

and we can write the body of our combine function in any of the following ways:

```scala
( sg.append(a,b), sg.append(a,c), sg.append(b,c) )

( a |+| b, a |+| c, b |+| c )

( a ⊹ b, a ⊹ c, b ⊹ c )

( a mappend b, a mappend c, b mappend c )
```

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



FP in Scala

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

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

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

```scala
def f[A](a: A, b: A, c: A)(implicit m: Monoid[A]): (A,A,A) =
  ( m.op(a, b), m.op(a, c), m.op(b, c) )
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```scala
trait Semigroup[F]  { self =>
  def append(f1: F, f2: => F): F
  …
```

```scala
trait Monoid[F] extends Semigroup[F] { self =>
  def zero: F
  …
```

```scala
final class SemigroupOps[F]…(implicit val F: Semigroup[F]) … {
  final def |+|(other: => F): F = F.append(self, other)
```

```scala
trait StringInstances {
  implicit object stringInstance extends Monoid[String] with …
  …
```

```scala
trait ListInstances extends ListInstances0 {
  …
  implicit def listMonoid[A]: Monoid[List[A]] = …
  …
```

```scala
trait AnyValInstances {
  …
  implicit val intInstance: Monoid[Int] with …
  …
```

```scala
import scalaz.Scalaz._
import scalaz._
```

```scala
def f[A](a: A, b: A, c: A)(implicit m: Monoid[A]): (A,A,A) =
  ( a |+| b, a |+| c, b |+| c )
```

```scala
assert( f("a","b","c") == ("ab","ac","bc"))
assert( f(List(1,2), List(3,4), List(5,6)) == (List(1, 2, 3, 4),List(1, 2, 5, 6),List(3, 4, 5, 6)) )
assert( f(1,2,3) == (3,4,5) )
```

Appendable Things

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

@typeclass trait Semigroup[A] {
  @op("|+|") def append(x: A, y: => A): A

  def multiply1(value: A, n: Int): A
}

@typeclass trait Monoid[A] extends Semigroup[A] {
  def zero: A

  def multiply(value: A, n:Int): A =
    if (n <= 0) zero else multiply1(value, n - 1)
}
```

|+| is known as the TIE Fighter operator. There is an Advanced TIE Fighter in an upcoming section, which is very exciting.

A **Semigroup** should exist for a type if two elements can be **combined** to produce another element of the same type. The operation must be **associative**, meaning that the order of nested operations should not matter, i.e.

```scala
(a |+| b) |+| c == a |+| (b |+| c)

(1 |+| 2) |+| 3 == 1 |+| (2 |+| 3)
```

A **Monoid** is a **Semigroup** with a **zero** element (also called **empty** or **identity**). Combining **zero** with any other a should give a.

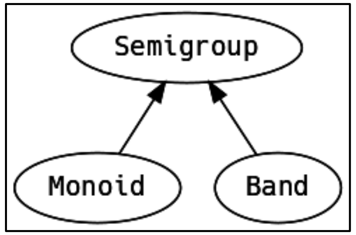
```scala
a |+| zero == a

a |+| 0 == a
```

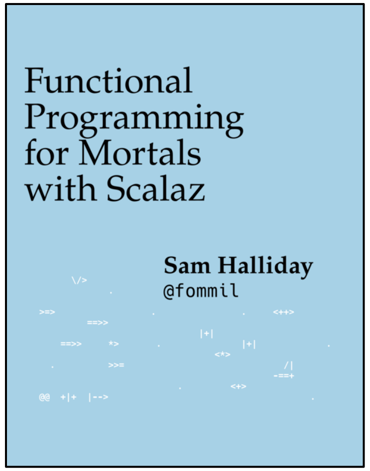
There are implementations of **Monoid** for all the primitive numbers, but the concept of **appendable** things is useful beyond numbers.

```scala
scala> "hello" |+| " " |+| "world!"
res: String = "hello world!"

scala> List(1, 2) |+| List(3, 4)
res: List[Int] = List(1, 2, 3, 4)
```

```scala
@typeclass trait Band[A] extends Semigroup[A]
```

Band has the law that the **append** operation of the same two elements is **idempotent**, i.e. gives the same value. Examples are **anything that can only be one value**, such as **Unit**, least upper bounds, or a **Set**. **Band** provides no further methods yet users can make use of the guarantees for performance optimisation.





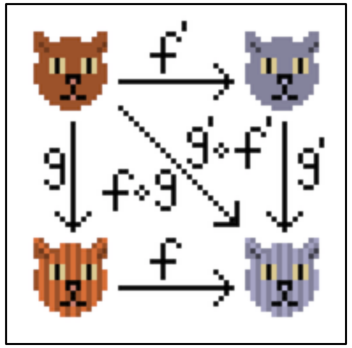
Sam Halliday

@fommil

Here is a simplified version of the **Monoid** definition from **Cats**

```scala
trait Monoid[A] {
  def combine(x: A, y: A): A
  def empty: A
}
```

In **Cats** the **binary operation** is called neither **op** nor **append**, but rather **combine** and the **identity** value is not called **zero** but **empty**.



by Noel Welsh and Dave Gurnell

@noelwelsh @davegurnell

In addition to providing the **combine** and **empty** operations, **monoids** must formally obey several **laws**. For all values x, y, and z, in A, combine must be **associative** and **empty** must be an **identity element**

```scala
def associativeLaw[A](x: A, y: A, z: A)(implicit m: Monoid[A]): Boolean =
{
  m.combine(x, m.combine(y, z)) == m.combine(m.combine(x, y), z)
}

def identityLaw[A](x: A)(implicit m: Monoid[A]): Boolean = {
  (m.combine(x, m.empty) == x) && (m.combine(m.empty, x) == x)
}
```

Integer subtraction, for example, is not a **monoid** because subtraction is not **associative**

```
scala> (1 - 2) - 3
res0: Int = -4

scala> 1 - (2 - 3)
res1: Int = 2
```

Cats

A **semigroup** is just the **combine** part of a **monoid**. While many **semigroups** are also **monoids**, **there are some data types for which we cannot define an empty element**. For example, we have just seen that sequence concatenation and integer addition are **monoids**. However, **if we restrict ourselves to non-empty sequences and positive integers, we are no longer able to define a sensible empty element. Cats has a NonEmptyList data type that has an implementation of Semigroup but no implementation of Monoid**.

```scala
import cats.Monoid
import cats.instances.int._
```

```
scala> Monoid[Int].combine(32, 10)
res4: Int = 42

scala> Monoid[Int].empty
res5: Int = 0
```

A more accurate (though still simplified) definition of **Cats' Monoid** is:

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

In **Cats**, as in **Scalaz**, the **binary operation** is defined in **Semigroup** rather than in **Monoid**.

```scala
import cats.Monoid
import cats.instances.string._
```

```
scala> Monoid[String].combine("Hi ", "there")
res2: String = Hi there

scala> Monoid[String].empty
res3: String = ""
```

As we know, **Monoid** extends **Semigroup**. If we don't need **empty** we can equivalently write:

```scala
import cats.Semigroup
import cats.instances.string._
```

```
scala> Semigroup[String].combine("Hi ", "there")
res6: String = Hi there
```

In **Cats** (as in **Scalaz**) **SemigroupOps** defines infix operator aliases for **Semigroup**'s associative operation, i.e. **combine** (**append**).

```
final class SemigroupOps[A: Semigroup](lhs: A) {
  def |+|(rhs: A): A = macro Ops.binop[A, A]
  def combine(rhs: A): A = macro Ops.binop[A, A]
  def combineN(rhs: Int): A = macro Ops.binop[A, A]
}
```

Given context and an expression, this method rewrites the tree to call the "desired" method with the **lhs** and **rhs** parameters.

by Noel Welsh and Dave Gurnell

@noelwelsh @davegurnell

```
import cats.Monoid
import cats.instances.string._ // for String Monoid
import cats.instances.int._    // for Int Monoid
```

```
scala> val stringResult = "Hi " combine "there" combine Monoid[String].empty
stringResult: String = Hi there


scala> val intResult = 1 combine 2 combine Monoid[Int].empty
intResult: Int = 3
```

Cats provides syntax for the **combine** method in the form of the **|+| operator**. Because **combine** technically comes from **Semigroup**, we access the syntax by importing from cats.syntax.semigroup

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

```
scala> val stringResult = "Hi " |+| "there" |+| Monoid[String].empty
stringResult: String = Hi there

scala> val intResult = 1 |+| 2 |+| Monoid[Int].empty
intResult: Int = 3
```

we saw the three infix operator aliases that **Scalaz** provides for **Semigroup**'s **append** function

@philip_schwarz

```
final class SemigroupOps[F]…(implicit val F: Semigroup[F]) … {
  final def |+|(other: => F): F      = F.append(self, other)
  final def mappend(other: => F): F = F.append(self, other)
  final def ⊹(other: => F): F       = F.append(self, other)
…
```

And we looked at **|+|**, aka the **TIE Fighter operator**.

What about **mappend**?

Monoid is an embarrassingly simple but amazingly powerful concept. It's the concept behind basic arithmetics: Both addition and multiplication form a monoid. **Monoids are ubiquitous in programming**. They show up as strings, lists, foldable data structures, futures in concurrent programming, events in functional reactive programming, and so on.
…

In **Haskell** we can define a type class for **monoids** — a type for which there is a **neutral element** called **mempty** and a **binary operation** called **mappend**:

```
class Monoid m where
mempty :: m
mappend :: m -> m -> m
```

…
As an example, let's declare **String** to be a **monoid** by providing the implementation of `mempty` and `mappend` (this is, in fact, done for you in the standard Prelude):

```
instance Monoid String where
mempty = ""
mappend = (++)
```

Here, we have reused the **list concatenation operator** (++), because a **String** is just a list of characters.

A word about **Haskell** syntax: Any infix operator can be turned into a two-argument function by surrounding it with parentheses. Given two strings, you can **concatenate** them by inserting ++ between them:

```
"Hello " ++ "world!”
```

or by passing them as two arguments to the parenthesized (++):

```
(++) "Hello " "world!"
```

@BartoszMilewski

In **Scalaz**, **mappend** is defined in **Semigroup**.

In **Haskell**, **mappend** is defined in **Monoid**.

Monoid

A monoid is a **binary associative operation** with an **identity**.

…

For **lists**, we have a **binary operator**, (++), that joins two lists together. We can also use a function, **mappend**, from the `Monoid` type class to do the same thing:

```
Prelude> mappend [1, 2, 3] [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

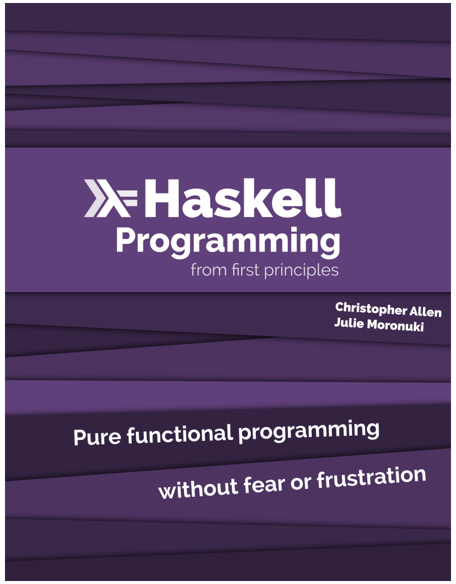
For **lists**, the empty list, [], is the **identity** value:

```
mappend [1..5] [] = [1..5]
mappend [] [1..5] = [1..5]
```

We can rewrite this as a more general rule, using **mempty** from the `Monoid` type class as a **generic identity value** (more on this later):

```
mappend x mempty = x
mappend mempty x = x
```

In plain English, **a monoid is a function that takes two arguments and follows two laws**: **associativity** and **identity**. **Associativity** means the arguments can be regrouped (or reparenthesized, or reassociated) in different orders and give the same result, as in addition. **Identity** means there exists some value such that when we pass it as input to our function, the operation is rendered moot and the other value is returned, such as when we add zero or multiply by one. **`Monoid` is the type class that generalizes these laws across types**.



By Christopher Allen
and Julie Moronuki

@bitemyapp @argumatronic

Again, in Haskell, **mappend** is defined in **Monoid**

The type class Monoid is defined:

```
class Monoid m where
mempty :: m
mappend :: m -> m -> m
mconcat :: [m] -> m
mconcat = foldr mappend mempty
```

mappend is **how any two values that inhabit your type can be joined together**. **mempty is the identity value for that mappend operation**. There are some laws that all Monoid instances must abide, and we'll get to those soon. Next, let's look at some examples of **monoids** in action!

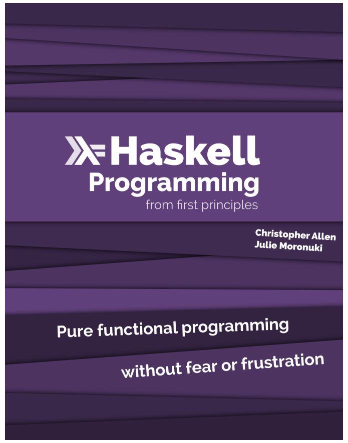
Examples of using Monoid

The nice thing about **monoids** is that they are familiar; they're all over the place. The best way to understand them initially is to look at examples of some common **monoidal** operations and remember that this type class abstracts the pattern out, giving you the ability to use the operations over a larger range of types.

List

One common type with an instance of **Monoid** is **List**. Check out how **monoidal** operations work with lists:

```
Prelude> mappend [1, 2, 3] [4, 5, 6]
[1,2,3,4,5,6]
Prelude> mconcat [[1..3], [4..6]]
[1,2,3,4,5,6]
Prelude> mappend "Trout" " goes well with garlic"
"Trout goes well with garlic"
```



**By Christopher Allen
and Julie Moronuki**
@bitemyapp @argumatronic

This should look familiar, because we've certainly seen this before:

```
Prelude> (++) [1, 2, 3] [4, 5, 6]
[1,2,3,4,5,6]
Prelude> (++) "Trout" " goes well with garlic"
"Trout goes well with garlic"
Prelude> foldr (++) [] [[1..3], [4..6]]
[1,2,3,4,5,6]
Prelude> foldr mappend mempty [[1..3], [4..6]]
[1,2,3,4,5,6]
```

Our old friend (++)! And if we look at the definition of Monoid for lists, we can see how this all lines up:

```
instance Monoid [a] where
mempty = []
mappend = (++)
```

For other types, the instances would be different, but the ideas behind them remain the same.

Semigroup

Mathematicians play with **algebras** like that creepy kid you knew in grade school who would pull legs off of insects. Sometimes, they glue legs onto insects too, but in the case where we're going from **Monoid** to **Semigroup**, we're pulling a leg off.

In this case, the leg is our **identity**. To get from a **monoid** to a **semigroup**, we simply no longer furnish nor require an **identity**. The **core operation** remains **binary** and **associative**. With this, our definition of **Semigroup** is:

```
class Semigroup a where
(<>) :: a -> a -> a
```

And we're left with one law:

```
(a <> b) <> c = a <> (b <> c)
```

Semigroup still provides a **binary associative operation**, one that typically **joins two things together** (as in **concatenation** or **summation**), but doesn't have an **identity** value. In that sense, it's a weaker **algebra**.

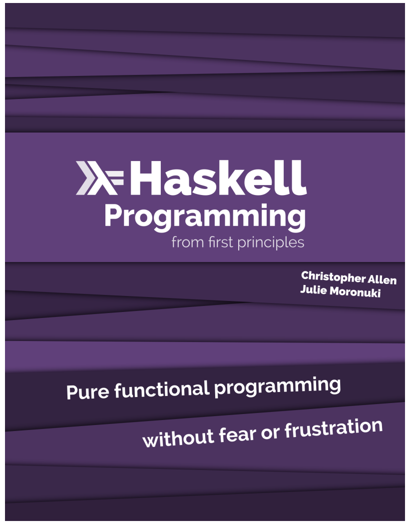
…

NonEmpty, a useful datatype

One useful datatype that can't have a **Monoid** instance but does have a **Semigroup** instance is the **NonEmpty** list type. It is a list datatype that can never be an empty list…

We can't write a **Monoid** for **NonEmpty** because it has no **identity** value by design! There is no empty list to serve as an **identity** for any operation over a **NonEmpty** list, yet there is still a **binary associative operation**: two **NonEmpty** lists can still be **concatenated**.

A type with a canonical **binary associative operation** but no **identity** value is a natural fit for **Semigroup**.



By Christopher Allen
and Julie Moronuki

@bitemyapp @argumatronic

@philip_schwarz

In **Scalaz** there is a predefined implicit **NonEmptyList Semigroup**

```
implicit def nonEmptyListSemigroup[A]: Semigroup[NonEmptyList[A]] = new Semigroup[NonEmptyList[A]] {
  def append(f1: NonEmptyList[A], f2: => NonEmptyList[A]) = f1 append f2
}
```

so if we write a function that operates on values of type A for which an **implicit Semigroup**, is available e.g. a function **foo** that appends two such values

```
def foo[A](x: A, y: A)(implicit sg: Semigroup[A]) =
  sg.append(x, y)
```

we are then able to use the function to append two non-empty lists

```
scala> foo( NonEmptyList(1,2,3), NonEmptyList(4,5,6) )
res2: scalaz.NonEmptyList[Int] = NonEmpty[1,2,3,4,5,6]
scala>
```

and since we saw before that there are infix operator aliases for the append method of a **Semigroup**, the body of **foo** can be written in any of the following ways

```
sg.append(x, y)
x |+| y
x ⊹ y
x mappend y
```

Strength can be weakness

When **Haskellers** talk about **the strength of an algebra**, they usually mean the number of operations it provides which in turn expands **what you can do with any given instance of that algebra without needing to know specifically what type you are working with**.

The reason we cannot and do not want to make all of our **algebras** as big as possible is that there are datatypes which are very useful representationally, but which do not have the ability to satisfy everything in a larger **algebra** that could work fine if you removed an operation or law.

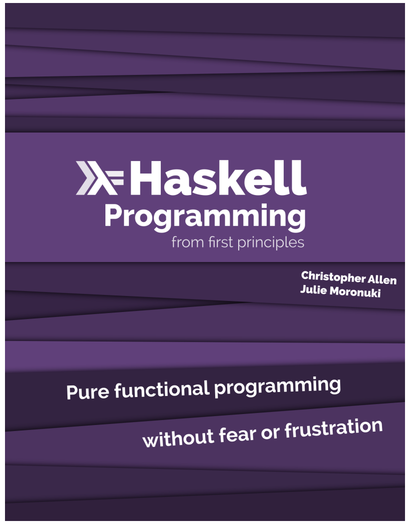
This becomes a serious problem if **NonEmpty** is the right datatype for something in the domain you're representing. If you're an experienced programmer, think carefully. **How many times have you meant for a list to never be empty? To guarantee this and make the types more informative, we use types like NonEmpty**.

The problem is that **NonEmpty has no identity** value for the **combining operation** (**mappend**) in **Monoid**. So, **we keep the associativity but drop the identity value and its laws of left and right identity**. **This is what introduces the need for and idea of Semigroup from a datatype**.

The most obvious way to see that **a monoid is stronger than a semigroup** is to observe that **it has a strict superset of the operations and laws that Semigroup provides**. **Anything which is a monoid is by definition also a semigroup**.

It is to be hoped that **Semigroup** will be made a **superclass** of **Monoid** in an upcoming version of GHC.

```
class Semigroup a => Monoid a where
...
```



By Christopher Allen
and Julie Moronuki

@bitemyapp @argumatronic

actually `Semigroup` *has* been made a superclass of `Monoid` – see next slide

Not Secure | hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:9

```
class Semigroup a => Monoid a where
```

The class of monoids (types with an associative binary operation that has an identity). Instances should satisfy the following laws:

- `x <> mempty = x`
- `mempty <> x = x`
- `x <> (y <> z) = (x <> y) <> z` (`Semigroup` law)
- `mconcat = foldr '(<>)' mempty`

The method names refer to the monoid of lists under concatenation, but there are many other instances.

Some types can be viewed as a monoid in more than one way, e.g. both addition and multiplication on numbers. In such cases we often define `newtype`s and make those instances of `Monoid`, e.g. `Sum` and `Product`.

NOTE: `Semigroup` is a superclass of `Monoid` since *base-4.11.0.0*.

Minimal complete definition

`mempty`

Methods

```
mempty :: a
```

Identity of `mappend`

```
mappend :: a -> a -> a
```

An associative operation

NOTE: This method is redundant and has the default implementation `mappend = '(<>)'` since *base-4.11.0.0*.

```
mconcat :: [a] -> a
```

Fold a list using the monoid.

For most types, the default definition for `mconcat` will be used, but the function is included in the class definition so that an optimized version can be provided for specific types.

So in Haskell, Monoid's mappend is actually just another name for Semigroup's associative operation <>, so maybe that's why in Scalaz, mappend is not defined in Monoid but is instead an infix operator that is an alias for Semigroup's associative function append.

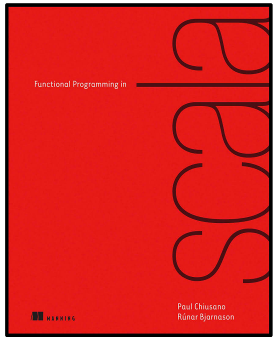
@philip_schwarz

The Option[A] Monoid and the notion that every Monoid has a dual

EXERCISE 10.1

Give a **Monoid** instance for combining **Option** values.

```
def optionMonoid[A]: Monoid[Option[A]]
```



FP in Scala

Notice that **we have a choice in how we implement op. We can compose the options in <u>either order</u>**.

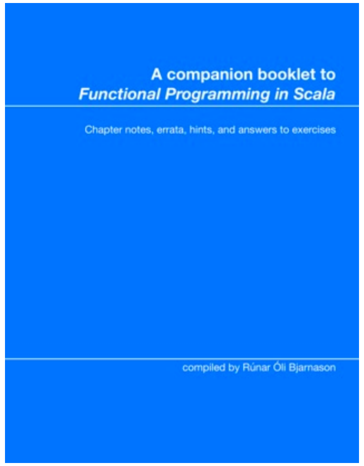
Both of those implementations satisfy the monoid laws, but they are not equivalent. This is true in general – that is, **every monoid has a <u>dual</u> where the op combines things in the <u>opposite order</u>**.

Monoids like **booleanOr** and **intAddition** are **equivalent to their <u>duals</u>** because their **op** is **commutative** as well as **associative**.

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

returns x if it is nonempty, otherwise returns the result of evaluating y

```
// We can get the dual of any monoid just by flipping the `op`.
def dual[A](m: Monoid[A]): Monoid[A] = new Monoid[A] {
  def op(x: A, y: A): A = m.op(y, x)
  val zero = m.zero
}

// Now we can have both monoids on hand:
def firstOptionMonoid[A]: Monoid[Option[A]] = optionMonoid[A]
def lastOptionMonoid[A]:  Monoid[Option[A]] = dual(firstOptionMonoid)
```



A Companion booklet to FP in Scala

```
scala> firstOptionMonoid.op(Some(2),Some(3))
res0: Option[Int] = Some(2)

scala> firstOptionMonoid.op(None,Some(3))
res1: Option[Int] = Some(3)

scala> firstOptionMonoid.op(Some(2),None)
res2: Option[Int] = Some(2)

scala> firstOptionMonoid.op(None,None)
res3: Option[Nothing] = None
```

```
scala> lastOptionMonoid.op(Some(2),Some(3))
res0: Option[Int] = Some(3)

scala> lastOptionMonoid.op(None,Some(3))
res1: Option[Int] = Some(3)

scala> lastOptionMonoid.op(Some(2),None)
res2: Option[Int] = Some(2)

scala> lastOptionMonoid.op(None,None)
res3: Option[Nothing] = None
```

The results of the op associative operations of firstOptionMonoid and lastOptionMonoid only differ when neither of the arguments is None.

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

Unlike the **op** of **monoids** like **booleanOr**, **booleanAnd**, **intAddition**, **intMultiplication**, which is **commutative**, the **op** of **monoids** like **stringMonoid** and **listMonoid** is **not commutative**, so these **monoids** are **not equivalent** to their **duals**.

```
scala> firstStringMonoid.op( "Hello, ", "World!" )
res0: String = Hello, World!

scala> lastStringMonoid.op( "Hello, ", "World!" )
res1: String = "World!Hello, "

scala> assert( firstStringMonoid.op("Hello, ", "World!") equals lastStringMonoid.op("World!", "Hello, ") )

scala>
```

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

```
scala> firstListMonoid[Int].op( List(1,2,3), List(4,5,6) )
res15: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> lastListMonoid[Int].op( List(1,2,3), List(4,5,6) )
res16: List[Int] = List(4, 5, 6, 1, 2, 3)
```

https://appdoc.app/artifact/org.scalaz/scalaz-core_2.9.2/7.0.0-RC2/scalaz/Tags$$Dual.html

scalaz.Tags

Dual

```
sealed trait Dual extends AnyRef
```

Type tag to choose a Monoid instance that inverts the operands to `append` before calling the natural Monoid for the type.

Example:

```
import scalaz.{@@, Tag, Tags, Dual}
import scalaz.std.string._
import scalaz.syntax.monoid._
import scalaz.Dual._
Dual("World!") |+| Dual("Hello, ") // "Hello, World!"
```

```
scala> "Hello, " |+| "World!"
res0: String = Hello, World!

scala> Dual("Hello, ") |+| Dual("World!")
res1: String @@ scalaz.Tags.Dual = "World!Hello, "

scala> Dual("World!") |+| Dual("Hello, ")
res2: String @@ scalaz.Tags.Dual = Hello, World!

scala> assert( ("Hello, " |+| "World!") equals (Dual("World!") |+| Dual("Hello, ")) )
```

```
scala> List(1,2,3) |+| List(4,5,6)
res3: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> Dual(List(1,2,3)) |+| Dual(List(4,5,6))
res4: List[Int] @@ scalaz.Tags.Dual = List(4, 5, 6, 1, 2, 3)
```

It looks like In **Scalaz** there is a **Dual tag** that we can apply to the operands of a **monoid**’s **associative operation** so that we get the same effect as using the **associative operation** of the **monoid**’s **dual**.

@philip_schwarz

SCALAZ

PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Using the **Dual tag** with the **String monoid**

and with the **List monoid**

The canonicity of a Scala monoid

In Scala, **it's possible to have multiple Monoid instances associated with a type**. For example, for the type Int, we can have a Monoid[Int] that uses addition with 0, and another Monoid[Int] that uses multiplication with 1.

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

```scala
val intMultiplication: Monoid[Int] = new Monoid[Int] {
    def op(x: Int, y: Int) = x * y
    val zero = 1
}
```

This can lead to certain problems since **we cannot count on a Monoid instance being canonical in any way**. To illustrate this problem, consider a "suspended" computation like the following:

```scala
case class Suspended(acc: Int, m: Monoid[Int], remaining: List[Int])
```

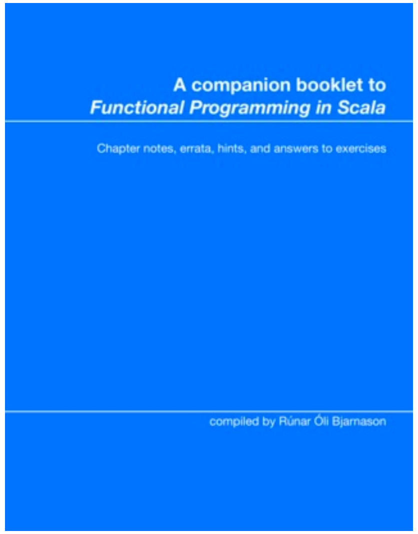
This represents an addition that is "in flight" in some sense. It's an accumulated value so far, represented by acc, a monoid m that was used to accumulate acc, and a list of remaining elements to add to the accumulation using the **monoid**.

Now, **if we have two values of type Suspended, how would we add them together?** We have no idea whether the two **monoids** are the same. And when it comes time to add the two acc values, which **monoid** should we use? There's no way of inspecting the **monoids** (since they are just functions) to see if they are equivalent. **So we have to make an arbitrary guess, or just give up.**

…

The **Scalaz** library takes the same approach [as **Haskell**], where **there is only one canonical monoid per type**. However, since **Scala** doesn't have type constraints, the **canonicity** of **monoids** is more of a convention than something enforced by the type system. And since **Scala** doesn't have newtypes, we use phantom types to add tags to the underlying types.

This is done with scalaz.**Tag**…



(by Runar Bjarnason)
@runarorama

There can only be one implementation of a typeclass for any given type parameter, a property known as **typeclass coherence**.

Typeclass coherence is primarily about **consistency**, and the **consistency** gives us the confidence to use implicit parameters. It would be difficult to reason about code that performs differently depending on the implicit imports that are in scope. **Typeclass coherence** effectively says that imports should not impact the behaviour of the code.

Tagging

In the section introducing **Monoid** we built a **Monoid**[TradeTemplate] and realised that scalaz does not do what we wanted with **Monoid**[**Option**[A]]. This is not an oversight of scalaz: **often we find that a data type can implement a fundamental typeclass in multiple valid ways and that the default implementation doesn't do what we want, or simply isn't defined**.

Basic examples are **Monoid**[**Boolean**] (**conjunction** && vs **disjunction** ||) and **Monoid**[**Int**] (**multiplication** vs **addition**).

To implement **Monoid**[**TradeTemplate**] we found ourselves either **breaking typeclass coherency**, or using a different typeclass.

scalaz.Tag **is designed to address the multiple typeclass implementation problem without breaking typeclass coherency**.

The definition is quite contorted, but the syntax to use it is very clean. This is how **we trick the compiler into allowing us to define an infix type A @@ T that is erased to A at runtime**:

…
<not shown here – too involved>
…

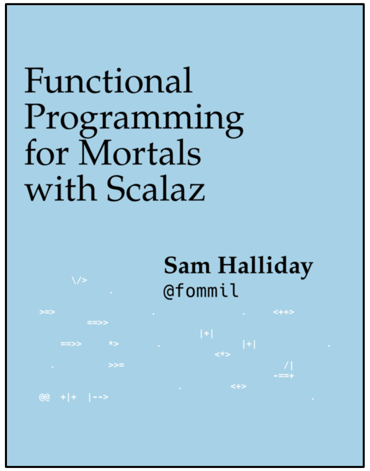
i.e. we tag things with **Princess Leia hair buns @@.**

Some useful tags are provided in the **Tags** object.

```
scala> import scalaz.Tags.{Disjunction,Multiplication}
import scalaz.Tags.{Disjunction, Multiplication}

scala> Multiplication(3)
res0: Int @@ scalaz.Tags.Multiplication = 3

scala> Disjunction(false)
res1: Boolean @@ scalaz.Tags.Disjunction = false
```

First / **Last** are used to select **Monoid** instances that pick the first or last non-zero operand. **Multiplication** is for numeric multiplication instead of addition. **Disjunction** / **Conjunction** are to select **&&** or **||**, respectively.

Using **scalaz.Tag** to distinguish between different **monoids** for the same type



Sam Halliday @fommil

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```
object Tags {

  sealed trait First
  val First = Tag.of[First]

  sealed trait Last
  val Last = Tag.of[Last]

  sealed trait Multiplication
  val Multiplication = Tag.of[Multiplication]

  sealed trait Disjunction
  val Disjunction = Tag.of[Disjunction]

  sealed trait Conjunction
  val Conjunction = Tag.of[Conjunction]

  ...
}
```

scalaz

Tags

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

`object` **Tags**

Type tags that are used to discriminate between alternative type class instances.

Source Tags.scala

See also scalaz.Tag and, @@ in the package object scalaz .

`trait` **Conjunction**

Type tag to choose a scalaz.Monoid instance that performs conjunction (&&)

`trait` **Disjunction**

Type tag to choose a scalaz.Monoid instance that performs disjunction (| |)

`trait` **First**

Type tag to choose a scalaz.Monoid instance that selects the first non-`zero` operand to append.

`trait` **Last**

Type tag to choose a scalaz.Monoid instance that selects the last non-`zero` operand to append.

`trait` **Multiplication**

Type tag to choose a scalaz.Monoid instance for a numeric type that performs multiplication, rather than the default monoid for these types which by convention performs addition.

Type Members

`sealed trait` **Conjunction**

Type tag to choose a scalaz.Monoid instance that performs conjunction (&&)

`sealed trait` **Disjunction**

Type tag to choose a scalaz.Monoid instance that performs disjunction (| |)

`sealed trait` **Dual**

Type tag to choose a scalaz.Monoid instance that inverts the operands to `append` before calling the natural scalaz.Monoid for the type.

`sealed trait` **First**

Type tag to choose a scalaz.Monoid instance that selects the first non-`zero` operand to append.

`sealed trait` **FirstVal**

Type tag to choose a scalaz.Semigroup instance that selects the first operand to append.

`sealed trait` **Last**

Type tag to choose a scalaz.Monoid instance that selects the last non-`zero` operand to append.

`sealed trait` **LastVal**

Type tag to choose a scalaz.Semigroup instance that selects the last operand to append.

`sealed trait` **Max**

Type tag to choose a scalaz.Monoid instance that selects the greater of two operands, ignoring `zero`.

`sealed trait` **MaxVal**

Type tag to choose a scalaz.Semigroup instance that selects the greater of two operands.

`sealed trait` **Min**

Type tag to choose a scalaz.Monoid instance that selects the lesser of two operands, ignoring `zero`.

`sealed trait` **MinVal**

Type tag to choose a scalaz.Semigroup instance that selects the lesser of two operands.

`sealed trait` **Multiplication**

Type tag to choose a scalaz.Monoid instance for a numeric type that performs multiplication, rather than the default monoid for these types which by convention performs addition.

`sealed trait` **Parallel**

Type tag to choose a scalaz.Applicative instance that runs scalaz.concurrent.Futures in parallel.

`sealed trait` **Zip**

Type tag to choose as scalaz.Applicative instance that performs zipping.

```
scala> // use default Scalaz Int monoid, i.e. (Int,+,0)

scala> 2 |+| 3
res0: Int = 5

scala> import scalaz.Tags.Multiplication
import scalaz.Tags.Multiplication

scala> // use alternative Scalaz Int monoid, i.e. (Int,*,1)

scala> Multiplication(2) |+| Multiplication(3)
res1: Int @@ scalaz.Tags.Multiplication = 6
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Examples of using **scalaz.Tag** to distinguish between different **Int monoids** and **Boolean monoids**

trait **Multiplication**

Type tag to choose a scalaz.Monoid instance for a numeric type that performs multiplication, rather than the default monoid for these types which by convention performs addition.

Princess Leia hair buns @@

```
scala> import scalaz.Scalaz._
import scalaz.Scalaz._
scala> import scalaz.Tags.{Conjunction,Disjunction}
import scalaz.Tags.{Conjunction, Disjunction}

scala> Conjunction(true)
res0: Boolean @@ scalaz.Tags.Conjunction = true
scala> Disjunction(true)
res1: Boolean @@ scalaz.Tags.Disjunction = true

scala> // use monoid (Boolean,OR,false)
scala> assert( (Disjunction(false) |+| Disjunction(false)) === Disjunction(false) )
scala> assert( (Disjunction(false) |+| Disjunction(true))  === Disjunction(true)  )
scala> assert( (Disjunction(true)  |+| Disjunction(false)) === Disjunction(true)  )
scala> assert( (Disjunction(true)  |+| Disjunction(true))  === Disjunction(true)  )

scala> // use monoid (Boolean,AND,true)
scala> assert( (Conjunction(false) |+| Conjunction(false)) === Conjunction(false) )
scala> assert( (Conjunction(false) |+| Conjunction(true))  === Conjunction(false) )
scala> assert( (Conjunction(true)  |+| Conjunction(false)) === Conjunction(false) )
scala> assert( (Conjunction(true)  |+| Conjunction(true))  === Conjunction(true)  )
```

trait **Disjunction**

Type tag to choose a scalaz.Monoid instance that performs disjunction (||)

trait **Conjunction**

Type tag to choose a scalaz.Monoid instance that performs conjunction (&&)

Picking a particular Boolean **semigroup** or **monoid** in **Scalaz**

There is a way of doing this, e.g. picking (**Boolean**, AND, **true**)

```
scala> import scalaz.Monoid
import scalaz.Monoid

scala> implicit val booleanMonoid: Monoid[Boolean] = scalaz.std.anyVal.booleanInstance.conjunction
booleanMonoid: scalaz.Monoid[Boolean] = scalaz.std.AnyValInstances$booleanInstance$conjunction$@4d2667fc

scala> import scalaz.syntax.semigroup._
import scalaz.syntax.semigroup._

scala> true |+| false
res0: Boolean = false

scala> booleanMonoid.zero
res3: Boolean = true
```

or picking (**Boolean**, OR, **false**)

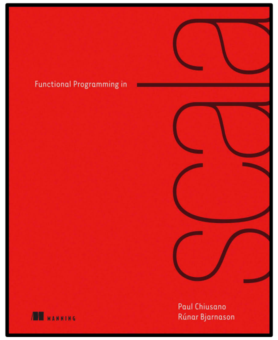
```
scala> import scalaz.Monoid
import scalaz.Monoid

scala> implicit val booleanMonoid: Monoid[Boolean] = scalaz.std.anyVal.booleanInstance.disjunction
booleanMonoid: scalaz.Monoid[Boolean] = scalaz.std.AnyValInstances$booleanInstance$disjunction$@794091e3

scala> import scalaz.syntax.semigroup._
import scalaz.syntax.semigroup._

scala> true |+| false
res0: Boolean = true

scala> booleanMonoid.zero
res3: Boolean = false
```

but as Travis Brown explains in his answer to https://stackoverflow.com/questions/34163121/how-to-create-semigroup-for-boolean-when-using-scalaz
this is somewhat at odds with the **Scalaz** philosophy

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

returns x if it is nonempty, otherwise returns the result of evaluating y

```scala
// We can get the dual of any monoid just by flipping the `op`.
def dual[A](m: Monoid[A]): Monoid[A] = new Monoid[A] {
  def op(x: A, y: A): A = m.op(y, x)
  val zero = m.zero
}

// Now we can have both monoids on hand:
def firstOptionMonoid[A]: Monoid[Option[A]] = optionMonoid[A]
def lastOptionMonoid[A]:  Monoid[Option[A]] = dual(firstOptionMonoid)
```



FP in Scala

Remember the two definitions of Monoid[Option[A]] we saw in **FP in Scala**, i.e. optionMonoid and its dual?

When firstOptionMonoid combines two Option arguments the result is the first non-zero argument, i.e. the first argument that is not None.

When lastOptionMonoid combines two Option arguments the result is the last non-zero argument, i.e. the last argument that is not None.

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

@philip_schwarz

In Scalaz, the above two Option monoids are called optionFirst and optionLast and are considered alternative Option monoids.

In Scalaz the default Option monoid is a third one called optionMonoid. It operates on Option[A] values such that a Semigroup[A] instance is defined.

When optionMonoid combines two Option arguments, the result is the result of combining the A values of the two options with the associative operation of the Semigroup[A] instance.

e.g. while the result of combining Some(2) and Some(3) with optionFirst is Some(2) and the result of combining them with optionLast is Some(3), the result of combining them with optionMonoid is Some(5), if Semigroup (Int,+) is chosen, or Some(6) if Semigroup (Int,*) is chosen.

```scala
implicit def optionMonoid[A: Semigroup]: Monoid[Option[A]] =
  new OptionSemigroup[A] with Monoid[Option[A]] {
    override def B = implicitly
    override def zero = None
  }

private trait OptionSemigroup[A] extends Semigroup[Option[A]] {
  def B: Semigroup[A]
  def append(a: Option[A], b: => Option[A]): Option[A] = (a, b) match {
    case (Some(aa), Some(bb)) => Some(B.append(aa, bb))
    case (Some(_), None) => a
    case (None, b2@Some(_)) => b2
    case (None, None) => None
  }
}
```

```
scala> Option(2) |+| Option(3)
res0: Option[Int] = Some(5)
scala> import scalaz.Tags.Multiplication
import scalaz.Tags.Multiplication
scala> Option(Multiplication(2)) |+| Option(Multiplication(3))
res1: Option[Int @@ scalaz.Tags.Multiplication] = Some(6)
```

Examples of `optionMonoid[A: Semigroup]: Monoid[Option[A]]` where A is `(Int,+)`, `(String,++)` and `(List[Int],++)`

gaining access to `|+|` using `Option(…)` and `None`

```
scala> Option(2) |+| Option(3)
res0: Option[Int] = Some(5)
scala> Option(2) |+| None
res1: Option[Int] = Some(2)
scala> (None:Option[Int]) |+| Option(3)
res2: Option[Int] = Some(3)

scala> Option("Hello, ") |+| Option("World!")
res3: Option[String] = Some(Hello, World!)
scala> Option("Hello, ") |+| None
res4: Option[String] = Some(Hello, )
scala> (None:Option[String]) |+| Option("World!")
res5: Option[String] = Some(World!)

scala> Option(List(1,2,3)) |+| Option(List(4,5))
res6: Option[List[Int]] = Some(List(1,2,3,4,5))
scala> Option(List(1,2,3)) |+| None
res7: Option[List[Int]] = Some(List(1,2,3))
scala> (None:Option[List[Int]]) |+| Option(List(1,2,3))
res8: Option[List[Int]] = Some(List(1,2,3))
```

using the more convenient **some** and **none** methods provided by **`OptionFunctions`**

```
scala> some(2) |+| some(3)
res0: Option[Int] = Some(5)
scala> some(2) |+| none
res1: Option[Int] = Some(2)
scala> none[Int] |+| some(3)
res2: Option[Int] = Some(3)

scala> some("Hello, ") |+| some("World!")
res3: Option[String] = Some(Hello, World!)
scala> some("Hello, ") |+| none
res4: Option[String] = Some(Hello, )
scala> none[String] |+| some("World!")
res5: Option[String] = Some(World!)

scala> some(List(1,2,3)) |+| some(List(4,5))
res6: Option[List[Int]] = Some(List(1,2,3,4,5))
scala> some(List(1,2,3)) |+| none
res7: Option[List[Int]] = Some(List(1,2,3))
scala> none[List[Int]] |+| some(List(1,2,3))
res8: Option[List[Int]] = Some(List(1,2,3))
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```
trait OptionFunctions {
  final def some[A](a: A): Option[A] = Some(a)
  final def none[A]: Option[A] = None
  …
```

Even more convenient: using the **some** method provided by **OptionIdOps**

```
scala> 2.some |+| 3.some
res0: Option[Int] = Some(5)
scala> 2.some |+| none
res1: Option[Int] = Some(2)
scala> none[Int] |+| 3.some
res2: Option[Int] = Some(3)

scala> "Hello, ".some |+| "World!".some
res3: Option[String] = Some(Hello, World!)
scala> "Hello, ".some |+| none
res4: Option[String] = Some(Hello, )
scala> none[String] |+| "World!".some
res5: Option[String] = Some(World!)

scala> List(1,2,3).some |+| List(4,5).some
res6: Option[List[Int]] = Some(List(1,2,3,4,5))
scala> List(1,2,3).some |+| none
res7: Option[List[Int]] = Some(List(1,2,3))
scala> none[List[Int]] |+| List(1,2,3).some
res8: Option[List[Int]] = Some(List(1,2,3))
```

```
final class OptionIdOps[A](val self: A) extends AnyVal {
  def some: Option[A] = Some(self)
}
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

How Scalaz alternative Option monoids optionFirst and optionLast are implemented using FirstOption[A] and LastOption[A], which are just aliases

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Choosing the optionFirst monoid or the optionLast monoid by using the First and Last tags

```
implicit def optionFirst[A]: Monoid[FirstOption[A]] with Band[FirstOption[A]] =
  new Monoid[FirstOption[A]] with Band[FirstOption[A]] {

    def zero: FirstOption[A] = Tag(None)

    def append(f1: FirstOption[A], f2: => FirstOption[A]) =
      Tag(Tag.unwrap(f1).orElse(Tag.unwrap(f2)))
}

implicit def optionLast[A]: Monoid[LastOption[A]] with Band[LastOption[A]] =
  new Monoid[LastOption[A]] with Band[LastOption[A]] {

    def zero: LastOption[A] = Tag(None)

    def append(f1: LastOption[A], f2: => LastOption[A]) =
      Tag(Tag.unwrap(f2).orElse(Tag.unwrap(f1)))
}
```

```
type FirstOption[A] = Option[A] @@ Tags.First
type LastOption[A] = Option[A] @@ Tags.Last
```

trait **First**

Type tag to choose a scalaz.Monoid instance that selects the first non-zero operand to append.

trait **Last**

Type tag to choose a scalaz.Monoid instance that selects the last non-zero operand to append.

```
scala> import scalaz.Tags.{First,Last}
import scalaz.Tags.{First, Last}

scala> First(2.some) |+| First(3.some)
res0: Option[Int] @@ scalaz.Tags.First = Some(2)
scala> First(2.some) |+| First(none)
res1: Option[Int] @@ scalaz.Tags.First = Some(2)
scala> First(none[Int]) |+| First(3.some)
res2: Option[Int] @@ scalaz.Tags.First = Some(3)
scala> First(none[Int]) |+| First(none)
res3: Option[Int] @@ scalaz.Tags.First = None

scala> Last(2.some) |+| Last(3.some)
res4: Option[Int] @@ scalaz.Tags.Last = Some(3)
scala> Last(2.some) |+| Last(none)
res5: Option[Int] @@ scalaz.Tags.Last = Some(2)
scala> Last(none[Int]) |+| Last(3.some)
res6: Option[Int] @@ scalaz.Tags.Last = Some(3)
scala> Last(none[Int]) |+| Last(none)
res7: Option[Int] @@ scalaz.Tags.Last = None
```

```
final class OptionOps[A](self: Option[A]) {
  …
  final def first: Option[A] @@ First = Tag(self)
  final def last: Option[A] @@ Last = Tag(self)
  …
```

Choosing the optionFirst monoid or the optionLast monoid by using the more convenient first and last methods provided by OptionOps

```
scala> 2.some.first |+| 3.some.first
res0: Option[Int] @@ scalaz.Tags.First = Some(2)
scala> 2.some.first |+| none.first
res1: Option[Int] @@ scalaz.Tags.First = Some(2)
scala> none[Int].first |+| 3.some.first
res2: Option[Int] @@ scalaz.Tags.First = Some(3)
scala> none[Int].first |+| none.first
res3: Option[Int] @@ scalaz.Tags.First = None

scala> 2.some.last |+| 3.some.last
res4: Option[Int] @@ scalaz.Tags.Last = Some(3)
scala> 2.some.last |+| none.last
res5: Option[Int] @@ scalaz.Tags.Last = Some(2)
scala> none[Int].last |+| 3.some.last
res6: Option[Int] @@ scalaz.Tags.Last = Some(3)
scala> none[Int].last |+| none.last
res7: Option[Int] @@ scalaz.Tags.Last = None
```

Same as in previous slide, but instead of looking at (Int,+) we look at (String,++) and (List[Int],++)

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

using the **First** and **Last** tags

```
scala> First("Hello, ".some) |+| First("World!".some)
res0: Option[String] @@ scalaz.Tags.First = Some(Hello, )
scala> First("Hello, ".some) |+| First(none)
res1: Option[String] @@ scalaz.Tags.First = Some(Hello, )
scala> First(none[String]) |+| First("World!".some)
res2: Option[String] @@ scalaz.Tags.First = Some(World!)
scala> First(none[String]) |+| First(none)
res3: Option[String] @@ scalaz.Tags.First = None

scala> Last("Hello, ".some) |+| Last("World!".some)
res4: Option[String] @@ scalaz.Tags.Last = Some(World!)
scala> Last("Hello, ".some) |+| Last(none)
res5: Option[String] @@ scalaz.Tags.Last = Some(Hello, )
scala> Last(none[String]) |+| Last("World!".some)
res6: Option[String] @@ scalaz.Tags.Last = Some(World!)
scala> Last(none[String]) |+| Last(none)
res7: Option[String] @@ scalaz.Tags.Last = None
```

```
scala> First(List(1,2,3).some) |+| First(List(4,5).some)
res0: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> First(List(1,2,3).some) |+| First(none)
res1: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> First(none[List[Int]]) |+| First(List(1,2,3).some)
res2: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> First(none[List[Int]]) |+| First(none)
res3: Option[List[Int]] @@ scalaz.Tags.First = None

scala> Last(List(1,2,3).some) |+| Last(List(4,5).some)
res4: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(4, 5))
scala> Last(List(1,2,3).some) |+| Last(none)
res5: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(1, 2, 3))
scala> Last(none[List[Int]]) |+| Last(List(1,2,3).some)
res6: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(1, 2, 3))
scala> Last(none[List[Int]]) |+| Last(none)
res7: Option[List[Int]] @@ scalaz.Tags.Last = None
```

using the more convenient **first** and **last** methods provided by **OptionOps**

```
scala> "Hello, ".some.first |+| "World!".some.first
res0: Option[String] @@ scalaz.Tags.First = Some(Hello, )
scala> "Hello, ".some.first |+| none.first
res1: Option[String] @@ scalaz.Tags.First = Some(Hello, )
scala> none[String].first |+| "World!".some.first
res2: Option[String] @@ scalaz.Tags.First = Some(World!)
scala> none[String].first |+| none.first
res3: Option[String] @@ scalaz.Tags.First = None

scala> "Hello, ".some.last |+| "World!".some.last
res4: Option[String] @@ scalaz.Tags.Last = Some(World!)
scala> "Hello, ".some.last |+| none.last
res5: Option[String] @@ scalaz.Tags.Last = Some(Hello, )
scala> none[String].last |+| "World!".some.last
res6 Option[String] @@ scalaz.Tags.Last = Some(World!)
scala> none[String].last |+| none.last
res7: Option[String] @@ scalaz.Tags.Last = None
```

```
scala> List(1,2,3).some.first |+| List(4,5).some.first
res0: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> List(1,2,3).some.first |+| none.first
res1: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> none[List[Int]].first |+| List(1,2,3).some.first
res2: Option[List[Int]] @@ scalaz.Tags.First = Some(List(1, 2, 3))
scala> none[List[Int]].first |+| none.first
res3: Option[List[Int]] @@ scalaz.Tags.First = None

scala> List(1,2,3).some.last |+| List(4,5).some.last
res4: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(4, 5))
scala> List(1,2,3).some.last |+| none.last
res5: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(1, 2, 3))
scala> none[List[Int]].last |+| List(1,2,3).some.last
res6: Option[List[Int]] @@ scalaz.Tags.Last = Some(List(1, 2, 3))
scala> none[List[Int]].last |+| none.last
res7: Option[List[Int]] @@ scalaz.Tags.Last = None
```

The Option Monoid in Cats

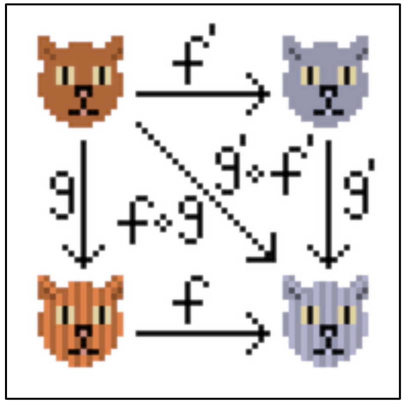
We saw earlier that in Scalaz there are three types of Option monoid: alternative monoids optionFirst and optionLast, plus a default one called optionMonoid, which operates on Option[A] values such that a Semigroup[A] instance is defined.
In Cats there is only one Option monoid and it has the same characteristics as the optionMonoid in Scalaz.

The `Option` monoid

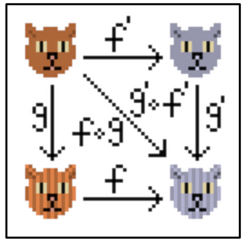
There are some types that can form a `Semigroup` but not a `Monoid`. For example, the following `NonEmptyList` type forms a semigroup through `++`, but has no corresponding identity element to form a monoid.

```scala
import cats.Semigroup

final case class NonEmptyList[A](head: A, tail: List[A]) {
  def ++(other: NonEmptyList[A]): NonEmptyList[A] = NonEmptyList(head, tail ++ other.toList)

  def toList: List[A] = head :: tail
}

object NonEmptyList {
  implicit def nonEmptyListSemigroup[A]: Semigroup[NonEmptyList[A]] =
    new Semigroup[NonEmptyList[A]] {
      def combine(x: NonEmptyList[A], y: NonEmptyList[A]): NonEmptyList[A] = x ++ y
    }
}
```

https://typelevel.org/cats/typeclasses/monoid.html



The **Cats** implementation of `optionMonoid[A: Semigroup]: Monoid[Option[A]]`



How then can we collapse a `List[NonEmptyList[A]]` ? For such types that only have a `Semigroup` we can lift into `Option` to get a `Monoid`.

```scala
import cats.syntax.semigroup._

implicit def optionMonoid[A: Semigroup]: Monoid[Option[A]] = new Monoid[Option[A]] {
  def empty: Option[A] = None

  def combine(x: Option[A], y: Option[A]): Option[A] =
    x match {
      case None => y
      case Some(xv) =>
        y match {
          case None => x
          case Some(yv) => Some(xv |+| yv)
        }
    }
}
```

https://typelevel.org/cats/typeclasses/monoid.html

This is the `Monoid` for `Option`: for any `Semigroup[A]`, there is a `Monoid[Option[A]]`.

```scala
implicit def optionMonoid[A: Semigroup]: Monoid[Option[A]] =
  new OptionSemigroup[A] with Monoid[Option[A]] {
    override def B = implicitly
    override def zero = None
  }

private trait OptionSemigroup[A] extends Semigroup[Option[A]] {
  def B: Semigroup[A]
  def append(a: Option[A], b: => Option[A]): Option[A] = (a, b) match {
    case (Some(aa), Some(bb)) => Some(B.append(aa, bb))
    case (Some(_), None) => a
    case (None, b2@Some(_)) => b2
    case (None, None) => None
  }
}
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

Comparing the **Cats** mplementation of **optionMonoid** with the **Scalaz** implementation.

Example of using the Option Monoid in Cats

Cats

We can assemble a **Monoid**[**Option**[**Int**]] using instances from `cats.instances.int` and `cats.instances.option`

```scala
import cats.Monoid
import cats.instances.int._    // for Monoid
import cats.instances.option._ // for Monoid

val a = Option(22)
// a: Option[Int] = Some(22)

val b = Option(20)
// b: Option[Int] = Some(20)

Monoid[Option[Int]].combine(a, b)
// res6: Option[Int] = Some(42)
```

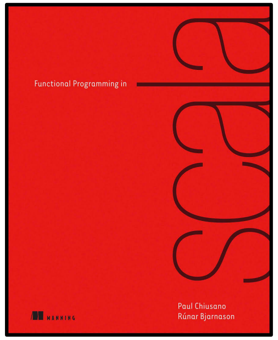
With the correct instances in scope, we can set about adding anything we want

```scala
import cats.instances.int._    // for Monoid
import cats.instances.option._ // for Monoid

Option(1) |+| Option(2)
// res1: Option[Int] = Some(3)
```

by Noel Welsh and Dave Gurnell

@noelwelsh @davegurnell

Summary of the naming and location of a **Monoid**'s **associative binary operation** and **identity element** - simplified



FP in Scala

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

```scala
trait Semigroup[F]  { self =>
  def append(f1: F, f2: => F): F
  …
}

final class SemigroupOps[F]…(implicit val F: Semigroup[F]) … {
  final def |+|(other: => F): F      = F.append(self, other)
  final def mappend(other: => F): F  = F.append(self, other)
  final def ⊹(other: => F): F        = F.append(self, other)
  …
}

trait Monoid[F] extends Semigroup[F] { self =>
  def zero: F
  …
}
```

```haskell
class Semigroup m where
(<>) :: m -> m -> m

class Semigroup m => Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

The mappend method is redundant and has the default implementation mappend = '(<>)'

Cats

```scala
trait Semigroup[A] {
  def combine(x: A, y: A): A
  …
}

final class SemigroupOps[A: Semigroup](lhs: A) {
  def |+|(rhs: A): A = macro Ops.binop[A, A]
  def combine(rhs: A): A = macro Ops.binop[A, A]
  def combineN(rhs: Int): A = macro Ops.binop[A, A]
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
  …
}
```

to be continued in part 2