MONOIDS

What, *How* and *Why*

Sebastian Galkin

@paraseba
paraseba@gmail.com

Scaladores - May 2018

OUTLINE

Ask questions as we go!

ABOUT ME

Sebastian Galkin

- Functional Programming for a while.
- Mostly big data and enterprise.
- You or your team want to learn more about FP?

e-mail: paraseba@gmail.com
twitter: @paraseba

WHAT IS ABSTRACTION?

Abstraction *is the process of extracting the underlying essence of a concept, removing any dependence on real world objects, and generalizing it so that it has wider applications.*

— Wikipedia, Abstraction (mathematics)

STANDING ON THE SHOULDERS OF GIANTS

Mathematicians are the masters of abstraction.

- Maximize generality.
- Maximize simplicity (elegance).
- Analogies and "analogies between analogies".
- Reuse whole theories.
- Find the right level of abstraction.

They have been doing this for centuries.

WHAT IS A MONOID?

Components - A triplet $(A, \bullet, u)$

- A [carrier] set (A)
- A binary operation ($\bullet$)
- An element of the set u

WHAT IS A MONOID?

Components - A triplet $(A, \bullet, u)$

- A [carrier] set (A)
- A binary operation $(\bullet)$
- An element of the set u

Laws $(a, b, c \in A)$

Closure: $a \bullet b$ is an element of A

Associativity: $(a \bullet b) \bullet c = a \bullet (b \bullet c)$

Identity: $u \bullet a = a \bullet u = a$

~~Commutativity:~~ $a \bullet b \neq b \bullet a$

We sometimes say "A has a monoid".

BASIC MONOIDS IN PROGRAMMING

In programming we identify the carrier set with a type.

A few monoids

- `Int` under `(+)` with `0`.
- `Int` under `(*)` with `1`.
- `Double` under `(+)` with `0`?

BASIC MONOIDS IN PROGRAMMING

In programming we identify the carrier set with a type.

A few monoids

- `Int` under `(+)` with `0`.
- `Int` under `(*)` with `1`.
- ~~`Double` under `(+)` with `0`~~ $(0.1 + 0.2) + 0.3 \neq 0.1 + (0.2 + 0.3)$.

BASIC MONOIDS IN PROGRAMMING

In programming we identify the carrier set with a type.

A few monoids

- `Int` under `(+)` with `0`.
- `Int` under `(*)` with `1`.
- ~~`Double` under `(+)` with `0`~~ $(0.1 + 0.2) + 0.3 \neq 0.1 + (0.2 + 0.3)$.
- `Boolean` under `||` with `False`.
- `Boolean` under `&&` with `True`.
- `Set[A]` under `union` with `Set.empty`.
- `Map[A,B]` under `(++)` with `Map.empty`.

There are many more.

MODELING MONOIDS IN SCALA

If the type `A` has a monoid we need:

- a way to combine two `A`s.
- a "special" value of type `A` for the identity.

MODELING MONOIDS IN SCALA

If the type `A` has a monoid we need:

- a way to combine two `A`s.
- a "special" value of type `A` for the identity.

```
trait Monoid[A] {

  // The associative operation (can't throw)
  def append(a: A, b: => A): A

  // The identity
  def zero: A
}
```

We can have more than one monoid for the same `A`

But wait

What happened to the laws?

OPTIONAL SYNTAX SUGAR

```
import MonoidSyntax._ // assume this in all slides

a |+| b === implicitly[Monoid[A]].append(a,b) // nicer syntax
```

No need to understand this code

```
object Monoid {
  def apply[A:Monoid]: Monoid[A] = implicitly[Monoid[A]]
}

object MonoidSyntax {
  implicit def ToMonoidOps[A:Monoid](v: A) = new MonoidOps[A](v)
  final class MonoidOps[A:Monoid](val self: A) {
    def |+|(other: => A): A = Monoid[A].append(self, other)
  }
}
```

TWO MONOIDS FOR `Int`

```
val intAddMon = new Monoid[Int] {

  def append(a: Int, b: => Int): Int =
    a + b

  def zero: Int = 0
}

val mulMon = new Monoid[Int] {

  def append(a: Int, b: => Int): Int =
    a * b

  def zero: Int = 1
}
```

Monoids instances are first class citizens.

THE LIST MONOID

```
List[Int](1, 2, 3, 4) |+| List[Int](5, 6, 7, 8) = List[Int](???)
```

THE LIST MONOID

List[Int](1, 2, 3, 4) |+| List[Int](5, 6, 7, 8) = List[Int](???)

A default (?) monoid for Lists

```
implicit def freeMon[A]: Monoid[List[A]] = new Monoid[List[A]] {

  def append(as: List[A], bs: => List[A]): List[A] =
    as ++ bs

  def zero: List[A] = List()
}
```

Our first noncommutative monoid.

MONOID FOR TUPLES

How can we combine two tuples `(A,B)`:

```
(4, List('H','e','l','l','o')) |+| (2, List('W','o','r','l','d'))
```

MONOID FOR TUPLES

How can we combine two tuples `(A,B)`:

```
(4, List('H','e','l','l','o')) |+| (2, List('W','o','r','l','d'))
```

If `A` and `B` have monoids themselves, we can combine componentwise.

```
implicit def pairMon[A: Monoid, B: Monoid] = new Monoid[(A, B)] {
  /*                          ↖ syntax sugar for:
     (implicit am: Monoid[A], implicit bm: Monoid[B]) */

  def append(x: (A, B), y: => (A, B)): (A,B) =
    (x._1 |+| y._1, x._2 |+| y._2)

  def zero: (A,B) =
    (Monoid[A].zero, Monoid[B].zero)
}
```

A MONOID FOR FUNCTIONS

```
Monoid[A => B]
```

A MONOID FOR FUNCTIONS

`Monoid[A => B]` where B has a monoid

Functions that return a monoidal type

```
implicit def monFunMon[A, B:Monoid]: Monoid[A => B] =
  new Monoid[A => B] {

    def zero: A => B =
      _ => Monoid[B].zero

    def append(f: A => B, g: => (A => B)): A => B =
      a => f(a) |+| g(a)
}
```

Convince yourself it is a lawful monoid.
Is it commutative?

RECAP

We have seen:

- Monoid definition (binary assoc operation with identity).
- Monoid `trait` (one type parameter, two methods: `zero`, `append`).
- Monoids for:
  - Numbers, lists, tuples (if components have monoids).
  - `A => M` where `M` has a monoid.
  - There are many, many more.

But why or how to use monoids.

A useful little function

```
// (a1 |+| a2) |+| a3 === a1 |+| (a2 |+| a3)
def mconcat[A:Monoid](as: Traversable[A]): A =
  as.foldLeft(Monoid[A].zero)(_ |+| _)
  // === as.foldRight(Monoid[A].zero)(_ |+| _)
```

A useful little function

```
// (a1 |+| a2) |+| a3 === a1 |+| (a2 |+| a3)
def mconcat[A:Monoid](as: Traversable[A]): A =
  as.foldLeft(Monoid[A].zero)(_ |+| _)
  // === as.foldRight(Monoid[A].zero)(_ |+| _)

def sum(xs: Traversable[Int]): Int =
  mconcat(xs)(intAddMon)

def concat[A](xs: Traversable[List[A]]): List[A] =
  mconcat(xs)(freeMon) // or get the Monoid implicitly
```

Super useful: `foldMap`

```
def foldMap[A, M:Monoid](as: Traversable[A], f: A => M): M =
  // === mconcat(as.map(f))

  as.foldLeft(Monoid[M].zero) { _ |+| f(_) }
```

`f` acts as the mapping phase in a MapReduce, the monoidal operation is the reduction step.

```
def filter[A](as: List[A], f: A => Boolean): List[A] =
  foldMap(as, (a:A) =>
    if (f(a)) List(a)
    else List())(freeMon)
```

COMPOSING MONOIDS

A monoid to compute min/max

A monoid for minimum

```
def minMon[A:Ordering]: Monoid[Option[A]] =
  new Monoid[Option[A]] {
```

COMPOSING MONOIDS

A monoid to compute min/max

A monoid for minimum

```
def minMon[A:Ordering]: Monoid[Option[A]] =
  new Monoid[Option[A]] {

    def append(a: Option[A], b: => Option[A]): Option[A] =
      (a, b) match {
        case (None, x) => x
        case (x, None) => x
        case (Some(x), Some(y)) => Some(Ordering[A].min(x,y))
      }

    def zero: Option[A] = None
}
```

We can do the same for `maxMon`.

COMPOSING MONOIDS

A monoid to compute min/max

Calculate `min` & `max` in a single pass

```
def min[A: Ordering](as: Traversable[A]): Option[A] =
  foldMap(as, (a:A) => Option(a))(minMon) // map and reduce
```

COMPOSING MONOIDS

A monoid to compute min/max

Calculate `min` & `max` in a single pass

```
def min[A: Ordering](as: Traversable[A]): Option[A] =
  foldMap(as, (a:A) => Option(a))(minMon) // map and reduce

def minMaxMon[A:Ordering] = ???

def minmax[A: Ordering](as: Traversable[A]): Option[(A,A)] =
```

COMPOSING MONOIDS

A monoid to compute min/max

Calculate min & max in a single pass

```
def min[A: Ordering](as: Traversable[A]): Option[A] =
  foldMap(as, (a:A) => Option(a))(minMon) // map and reduce

def minMaxMon[A:Ordering] = ???

def minmax[A: Ordering](as: Traversable[A]): Option[(A,A)] =
  foldMap(as, (a:A) => (Option(a), Option(a)))(minMaxMon) match {

    case (Some(mi), Some(ma)) => Some((mi,ma))
    case _ => None
  }
```

COMPOSING MONOIDS

A monoid to compute min/max

Calculate min & max in a single pass

```
def min[A: Ordering](as: Traversable[A]): Option[A] =
  foldMap(as, (a:A) => Option(a))(minMon) // map and reduce

def minMaxMon[A:Ordering] = pairMon(minMon, maxMon)

def minmax[A: Ordering](as: Traversable[A]): Option[(A,A)] =
  foldMap(as, (a:A) => (Option(a), Option(a)))(minMaxMon) match {

    case (Some(mi), Some(ma)) => Some((mi,ma))
    case _ => None
  }
```

- Note how easily the monoids compose.
- Can be extended to other aggregates (`min`, `max`, `sum`, `size`).

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`
- `Option[B]: Monoid`

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`
- `Option[B]: Monoid`
- `(A, Option[B]): Monoid`
- `List[(A, Option[B])]: Monoid`

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`
- `Option[B]: Monoid`
- `(A, Option[B]): Monoid`
- `List[(A, Option[B])]: Monoid`
- `Y => List[(A, Option[B])]: Monoid`

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`
- `Option[B]: Monoid`
- `(A, Option[B]): Monoid`
- `List[(A, Option[B])]: Monoid`
- `Y => List[(A, Option[B])]: Monoid`
- `X => Y => List[(A, Option[B])]: Monoid`
- `Map[String, X => Y => List[(A, Option[B])]]: Monoid`

Monoids are very expressive.

POWERFUL ABSTRACTION

Monoids compose extremely well.

- `A: Monoid`
- `B: Monoid`
- `Option[B]: Monoid`
- `(A, Option[B]): Monoid`
- `List[(A, Option[B])]: Monoid`
- `Y => List[(A, Option[B])]: Monoid`
- `X => Y => List[(A, Option[B])]: Monoid`
- `Map[String, X => Y => List[(A, Option[B])]]: Monoid`

Monoids are very expressive.

- `mconcat:  sum, concat, ...`
- `foldMap:  fold*, filter, map, ...`

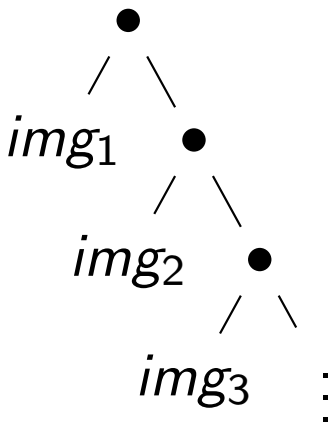
Signs of a good abstraction

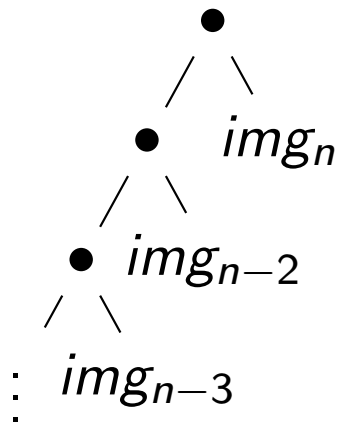
The problem

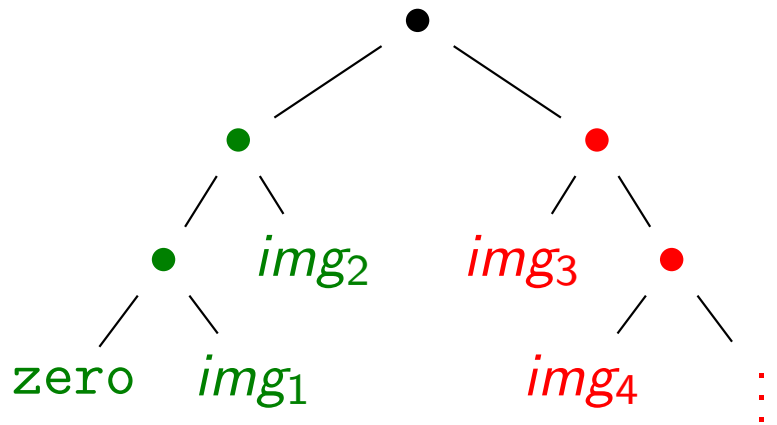
- Merge multiple renders into one.
- `merge` is an expensive operation.
- `merge(img1, merge(img2, merge(img3, ...)))`
- Obvious solution: `mconcat( List(img1, img2, ..., imgn) )`

The problem

- Merge multiple renders into one.
- `merge` is an expensive operation.
- `merge(img1, merge(img2, merge(img3, ...)))`
- Obvious solution: `mconcat( List(img1, img2, ..., imgn) )`







LARGER USE CASES

Parallelization

```
object Parallel {

  def mconcat[A:Monoid](as: Traversable[A]): A =
    as.foldLeft(Monoid[A].zero)(_ |+| _)
}
```

LARGER USE CASES

Parallelization

```
object Parallel {

  def mconcat[A:Monoid](as: Traversable[A]): A =
    as.fold(Monoid[A].zero)(_ |+| _)
}
```

```
fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

z: a neutral element for the fold operation; may be added to the result an arbitrary number of times, and must not change the result.
op: a binary operator that must be associative.

That's just a monoid!

LARGER USE CASES

Parallelization

```
object Parallel {

  def mconcat[A:Monoid](as: Traversable[A]): A =
    as.par.fold(Monoid[A].zero)(_ |+| _)
}
```

```
fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

z: a neutral element for the fold operation; may be added to the result an arbitrary number of times, and must not change the result.
op: a binary operator that must be associative.

That's just a monoid!

The problem

- System measures request latency.
- compute and store mean and variance.
- Events arrive at high rate, millions per hour.

The problem

- System measures request latency.
- compute and store mean and variance.
- Events arrive at high rate, millions per hour.

Naive approach

- Store all measurements,
- compute μ, σ^2 on demand, or precompute at certain interval.

Too much space, too much time. $\mathcal{O}(N)$.

LARGER USE CASES

Incremental updates

A better approach

- Find a monoid to combine mean/variance of n samples.
- Store only the stats, not the measurements.
- Accumulate several (or 1) new measurements.
- Combine the old and new stats using the monoid.
- Store.

How is it better?

- Exact if we can find the right monoid.
- $\mathcal{O}(1)$ updates (independent of sample size).
- Trade off freshness by scale.

What does the monoid need?

Weighted average of the means

$$n \cdot \mu = \sum_{i=0}^{n-1} x_i \qquad m \cdot \nu = \sum_{i=0}^{m-1} x_{i+n}$$

$$\frac{1}{n+m} \sum_{i=0}^{n+m-1} x_i = \frac{n \cdot \mu + m \cdot \nu}{n+m}$$

We need previous mean and sample size.
Generalizing, to compute the nth-moment, we need to store n+1 values.

A data type for mean and variance

```
sealed abstract class MeanVar

final case object EmptyMeanVar extends MeanVar

final case class MeanVarV(
  m1: Double,
  m2: Double,
  n: Long) extends MeanVar
```

The MeanVar monoid

```
val meanVarMonoid: Monoid[MeanVar] = new Monoid[MeanVar] {
  def zero: MeanVar = EmptyMeanVar

  def append(a: MeanVar, b: => MeanVar): MeanVar = (a, b) match {
    case (EmptyMeanVar, a) => a
    case (a, EmptyMeanVar) => a
```

The MeanVar monoid

```
val meanVarMonoid: Monoid[MeanVar] = new Monoid[MeanVar] {
  def zero: MeanVar = EmptyMeanVar

  def append(a: MeanVar, b: => MeanVar): MeanVar = (a, b) match {
    case (EmptyMeanVar, a) => a
    case (a, EmptyMeanVar) => a

    case (MeanVarV(m1a, m2a, na), MeanVarV(m1b, m2b, nb)) => {
      val (nt, delta) = (na + nb, m1b - m1a)
      MeanVarV((na * m1a + nb * m1b ) / nt,
               m2a + m2b + delta * delta * na * nb / nt,
               nt)
    }
  }
```

Utility functions: creating MeanVars

```
object MeanVar {

  def singleton(x: Double): MeanVar = MeanVarV(x, 0, 1)

  def sample(xs: Traversable[Double]): MeanVar =
```

Utility functions: creating MeanVars

```
object MeanVar {

  def singleton(x: Double): MeanVar = MeanVarV(x, 0, 1)

  def sample(xs: Traversable[Double]): MeanVar =
    foldMap(xs, singleton) // === mconcat(xs.map(singleton))
}
```

Updating the statistics (pseudocode)

```
// Define a time resolution (accumulate for 1sec? 100ms?)
val newSample: Vector[Double] = getNewMeasurements(...)

// Compute mean/var of the new data (incremental work)
val newStats: MeanVar = MeanVar.sample(newSample)

// This is an O(1) op
// In our example this loads 3 numbers,
// doesn't matter how many samples we have processed before
val oldStats: MeanVar = loadStats(...)

// Another O(1) operation
storeStats(oldStats |+| newStats)
```

LARGER USE CASES

Incremental updates

Notice:

- We never wrote the alg. to compute mean/var, only to combine them.
- Extensible to higher momenta.
- Extensible to approximate histograms and other fun stuff.
- Didn't we say `Double` under `(+)` is not a monoid?

CONCLUSIONS

- You could have invented the monoid.
- You could have discovered the laws.
- Write only lawful monoids.
- Use the monoid. They are everywhere.
- Don't be afraid of other algebraic structures (Functor, Monad).
- Learn how to abstract, copy and steal.

REFERENCES

- This talk: slides, all the code and many tests

  https://github.com/paraseba/scaladores-may-2018-talk
- *Functional Programming in Scala*
  Paul Chiusano & Runar Bjarnason
- *Why Functional Programming Matters*
  John Hughes.
- Scalaz library

  https://github.com/scalaz/scalaz
- *Computing skewness and kurtosis in one pass*

  https://www.johndcook.com/blog/skewness_kurtosis/