DESIGNING WITH MONOIDS (Part II)

Sebastian Galkin

@paraseba
paraseba@gmail.com

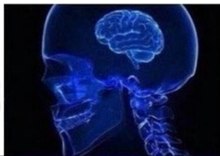
Scaladores - Nov 2018

Part I: https://github.com/paraseba/scaladores-may-2018-talk

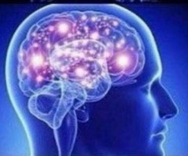
Ask questions!

We can go back and forth between slides.

MONOIDS FOR
SHORTER CODE



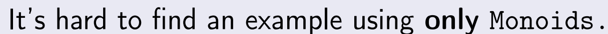
MONOIDS
FOR CODE REUSE



MONOIDS
AT THE CORE
OF THE PROBLEM

MONOIDS
TO GUIDE
THE DESIGN

imgflip.com



It's hard to find an example using **only** `Monoids`.

Brent A. Yorgey, "**Monoids: Theme and Variations (Functional Perl)**". July 2012.

PREREQUISITES

Find much more on the first part of this talk.

Semigroup and Monoid

```
trait Semigroup[M] {
  // moved from Monoid to its own class
  def append(a: M, b: => M): M
}

trait Monoid[M] extends Semigroup[M] {
  def zero: M
}
```

The Laws

- `append(a, append(b, c)) === append(append(a, b), c)`
- `append(a, zero) === append(zero, a) === a`

PREREQUISITES

Syntax

```
object SemigroupSyntax {
  final class SemigroupOps[M:Semigroup](val self: M) {
    def |+|(other: => M): M =
      Semigroup[M].append(self, other)
  }
```

The Canonical Examples

- (Int, +, 0)
- (Int, *, 1)
- (List[A], ++, [])
- (A => A, andThen, identity)

Using Monoids

```
implicit val intSumMon: Monoid[Int] = ....
implicit def listMon[A]: Monoid[List[A]] = ....

val five = 3 |+| 2
val l = List(1,2,3) |+| List(4,5,6)
```

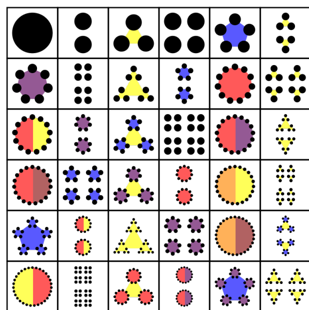
PREREQUISITES

The All-Purpose Function: `mconcat`

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

val five = mconcat(List(0,1,2,2))(intAddMonoid)
```

Any doubts about monoids?

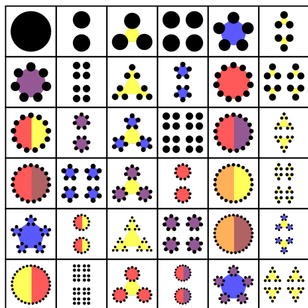
DIAGRAMS 1.0 — A LIST OF PRIMITIVES



A Diagram as a List of Primitives

```
sealed trait Prim { def draw(...) ... }

final case class Point(
  x: Double,
  y: Double,
  c: Color, ...) extends Prim

final case class Line(...) extends Prim
final case class Ellipse(...) extends Prim
...
```

DIAGRAMS 1.0 — A LIST OF PRIMITIVES



A Diagram as a List of Primitives

```
sealed trait Prim { def draw(...) ... }

final case class Point(
  x: Double,
  y: Double,
  c: Color, ...) extends Prim

final case class Line(...) extends Prim
final case class Ellipse(...) extends Prim
...

// Primitives are drawn in the list order
final case class Diagram(
  primitives: List[Prim])
```

How do Diagrams combine or compose?

DIAGRAMS 1.0 — A LIST OF PRIMITIVES

Diagrams compose by drawing them on top of each other.

The Diagram Monoid

```
final case class Diagram(primitives: List[Prim])

object Diagram {

  val diagramMonoid = new Monoid[Diagram] {
    def zero: Diagram = Diagram(listMon.zero)

    def append(d1: Diagram, d2: => Diagram): Diagram =
      Diagram(listMon.append(d1.primitives, d2.primitives))
  }
}
```

DIAGRAMS 1.0 — A LIST OF PRIMITIVES

Helper for Wrapped Monoids

```
def wMon[Out, In:Monoid](un: Out => In,
                         wr: In => Out): Monoid[Out] =
  new Monoid[Out] {
    def append(x: Out, y: => Out) = wr(un(x) |+| un(y))
    def zero = wr(implicitly[Monoid[In]].zero)
  }

def wSemi[Out, In:Semigroup](un: Out => In,
                             wr: In => Out): Semigroup[Out] =
  new Semigroup[Out] {
    def append(x:Out, y: => Out): Out = wr(un(x) |+| un(y))
  }
```

Using the implicit `Monoid` instance.

DIAGRAMS 1.0 — A LIST OF PRIMITIVES

The Diagram Monoid

```
// remember the definition of Diagram
final case class Diagram(primitives: List[Prim])

object Diagram {

  val diagramMonoid: Monoid[Diagram] =
    wMon(_.primitives, Diagram(_))
}
```

And now we can combine `Diagrams` drawing one on top of the other.
But we would want `d1 |+| d2` to draw `d1` on top.

DIAGRAMS 1.0 — A LIST OF PRIMITIVES

Fixing the Order

```
final case class Dual[M](undual: M)

implicit def dualMonoid[M:Monoid] = new Monoid[Dual[M]] {
  def zero = Dual(implicitly[Monoid[M]].zero)

  def append(a: Dual[M], b : => Dual[M]) =
    Dual(b.undual |+| a.undual)
}
```

DIAGRAMS 1.0 — A LIST OF PRIMITIVES

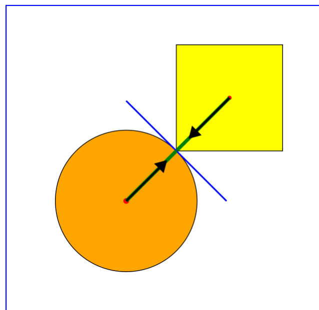
Fixing the Order

```
final case class Dual[M](undual: M)

implicit def dualMonoid[M:Monoid] = new Monoid[Dual[M]] {
  def zero = Dual(implicitly[Monoid[M]].zero)

  def append(a: Dual[M], b : => Dual[M]) =
    Dual(b.undual |+| a.undual)
}

final case class Diagram(primitives: Dual[List[Prim]])

object Diagram {
  val diaMon: Monoid[Diagram] = wMon(_.primitives, Diagram(_))
}
```

Look how expressive the type for `Diagram` is!

DIAGRAMS 2.0 — BESIDE

A different way to compose `Diagrams`.

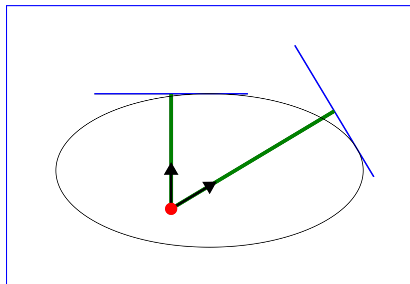


Monoids: Theme and Variations

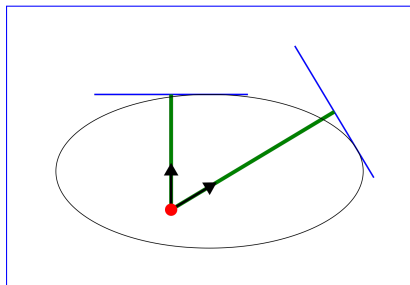
The beside Function

```
// vectors on the plane
type V2 = (Double, Double)

def beside(v: V2, d1: Diagram, d2: Diagram): Diagram = ???
```

DIAGRAMS 2.0 — BESIDE



DIAGRAMS 2.0 — BESIDE



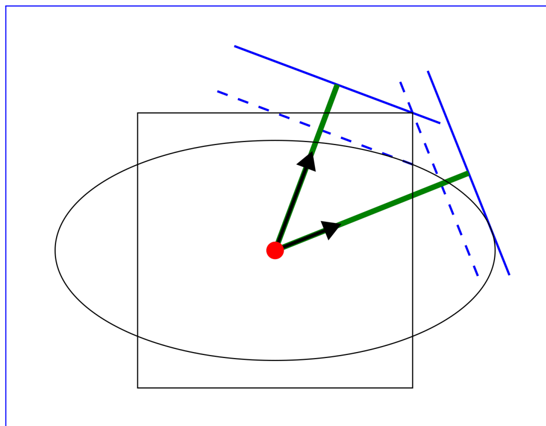
Monoids: Theme and Variations

Envelopes: Function as Data

```
final case class Envelope(f: V2 => Double)

def envelopeP(p: Prim): Envelope = ...
```

How do `Envelopes` compose?

DIAGRAMS 2.0 — BESIDE



Monoids: Theme and Variations

`Envelopes` compose by taking the maximum distance in every direction.

DIAGRAMS 2.0 — BESIDE

Reminder: Functions that return a Semigroup

```
implicit def funSemi[A, B:Semigroup]: Semigroup[A => B] =
  new Semigroup[A => B] {

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

We can use this for our

```
final case class Envelope(f:  V2 => Double)
```

DIAGRAMS 2.0 — BESIDE

A Semigroup for Envelopes

```scala
// libraries provide this type out of the box
final case class Max[A](unmax: A)

implicit def maxSemi[A:Ordering] = new Semigroup[Max[A]] {
  def append(a: Max[A], b: => Max[A]): Max[A] =
    Max(Ordering[A].max(a.unmax, b.unmax))
}
```

DIAGRAMS 2.0 — BESIDE

A Semigroup for Envelopes

```
// libraries provide this type out of the box
final case class Max[A](unmax: A)

implicit def maxSemi[A:Ordering] = new Semigroup[Max[A]] {
  def append(a: Max[A], b: => Max[A]): Max[A] =
    Max(Ordering[A].max(a.unmax, b.unmax))
}

// final case class Envelope(f: V2 => Double)
final case class Envelope(f: V2 => Max[Double])

val semiGroup: Semigroup[Envelope] = wSemi(_.f, Envelope(_))
```

But we push for a `Monoid` for `Envelope`.

DIAGRAMS 2.0 — BESIDE

Reminder: Option Monoid

```
// libraries provide this type out of the box
implicit def optionMon[A:Semigroup] = new Monoid[Option[A]] {

    def zero: Option[A] = None

    def append(a: Option[A], b: => Option[A]): Option[A] =
      (a,b) match {
        case (a, None) => a
        case (None, b) => b
        case (Some(a), Some(b)) => Some(a |+| b)
      }
}
```

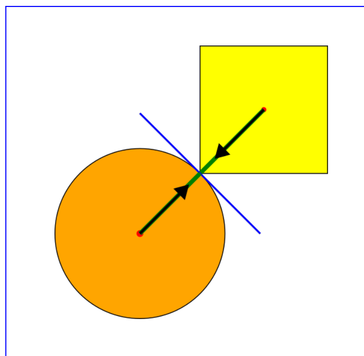
DIAGRAMS 2.0 — BESIDE

A Monoid for Envelopes

```
final case class Envelope(f: Option[V2 => Max[Double]])

val envMonoid: Monoid[Envelope] = wMon(_.f, Envelope(_))
```

Again, full expressiveness for `Envelope` type!

Now we can write `beside`



Monoids: Theme and Variations

Translate the square in the direction `v` by:
(Distance to the end of the circle in the direction `v`) +
(Distance to the end of the square in the direction `-v`)

DIAGRAMS 2.0 — BESIDE

beside Is Just Composition of Translated Diagrams

```
// Monoid magic
def envelope(d: Diagram): Envelope =
  mconcat(d.primitives.undual.map(envelopeP))

def translate(v: V2, d: Diagram): Diagram =
  Diagram(Dual(d.primitives.undual.map(translateP(v, _))))

def beside(v: V2, d1: Diagram, d2: Diagram): Diagram =
  (envelope(d1).f, envelope(d2).f) match {

    case (Some(f1), Some(f2)) =>
      d1 |+| translate(v * (f1(v).unmax + f2(-v).unmax), d2)

    case _ => d1 |+| d2
  }
```

DIAGRAMS 3.0 — CACHING

Computing `Envelopes` can become expensive.

Cached `Diagrams`

```
final case class Diagram private (
  primitives: Dual[List[Prim]], env: Envelope)

def envelope(d: Diagram): Envelope = d.env

def primDiag(p: Prim): Diagram =
  Diagram(Dual(List(p)), envelopeP(p))

object Diagram {
  def mkDiag(ps: List[Prim]): Diagram = mconcat(ps.map(primDiag))

  // using the tuple Monoid
  implicit val monoid: Monoid[Diagram] = wMon(..., ...)
}
```

DIAGRAMS 3.0 — CACHING

The only way to create composite diagrams is to use |+|

Complex Diagrams

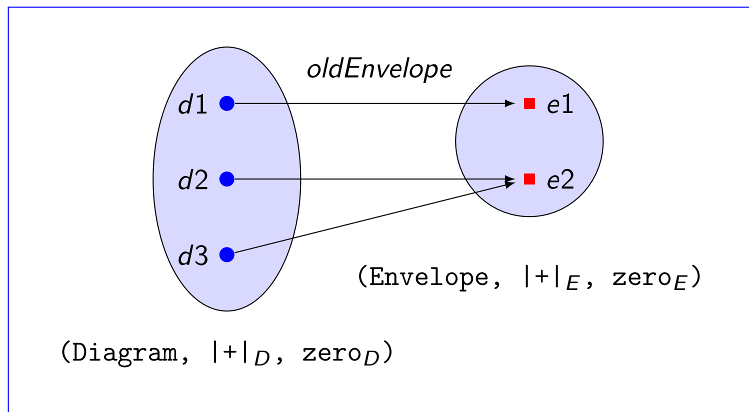
```
val d: Diagram = d1 |+| d2 |+| beside(v, d3, d4) |+| ...
```

But will this work?

We Need to Verify

```
def oldEnvelope(d: Diagram): Envelope =
  mconcat(d.primitives.undual.map(envelopeP))

oldEnvelope(Diagram(Dual(ps), e)) ==? e
```

In pictures, the `oldEnvelope` function:



DIAGRAMS 3.0 — CACHING

```
oldEnvelope(Diagram(Dual(ps), e)) ==?  e
```

Proof

oldEnvelope($zero_D$) $\longleftarrow$

$zero_D$ = mkDiag(List()) = Diagram(Dual(List.empty), $zero_E$)

oldEnvelope(mkDiag(List())) = mconcat(List.empty) = $zero_E$ $\longleftarrow$

DIAGRAMS 3.0 — CACHING

```
oldEnvelope(Diagram(Dual(ps), e)) ==?  e
```

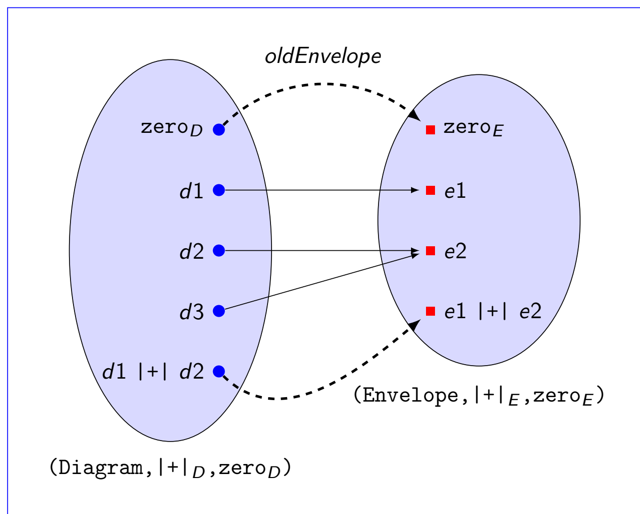
Proof

```
oldEnvelope(zero_D)                                                    ←
zero_D = mkDiag(List()) = Diagram(Dual(List.empty), zero_E)
oldEnvelope(mkDiag(List())) = mconcat(List.empty) = zero_E             ←



oldEnvelope(d1 |+|_D d2)   ←
oldEnvelope(Diagram(Dual(p2 ++ p1), e1 |+|_E e2))
= mconcat((p2 ++ p1).map(envelopeP))
= mconcat(p2.map(envelopeP) ++ p1.map(envelopeP))
= mconcat(p2.map(envelopeP)) |+|_E mconcat(p1.map(envelopeP))
= oldEnvelope(d2) |+|_E oldEnvelope(d1)
= oldEnvelope(d1) |+|_E oldEnvelope(d2)
= e1 |+|_E e2              ←
```

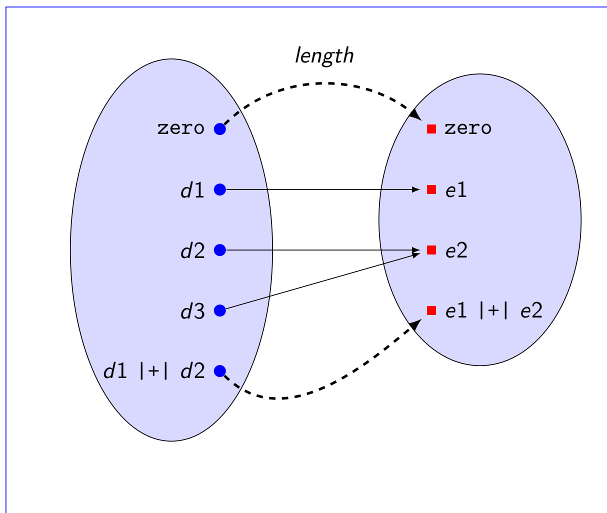
Equational reasoning!

DIAGRAMS 3.0 — CACHING

As it turns out, `oldEnvelope` is a `Monoid` homomorphism.

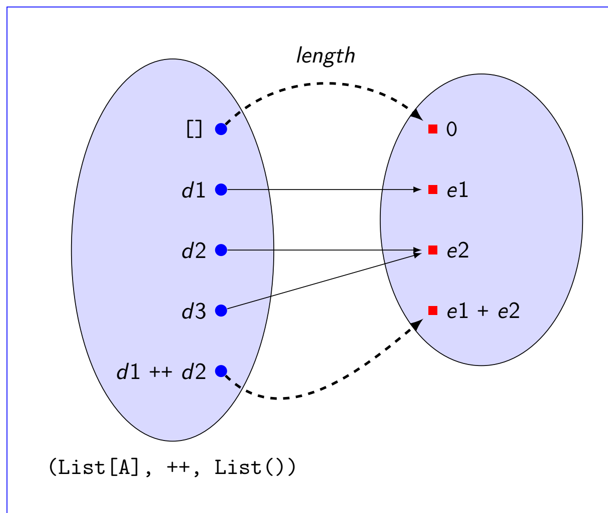


A simpler example of a `Monoid` homomorphism.

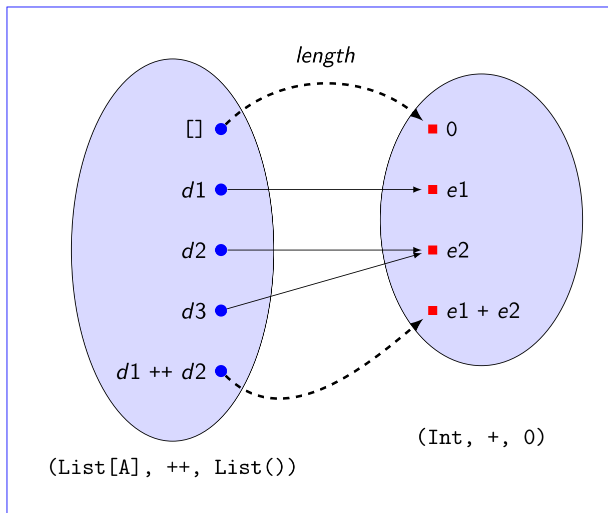


DIAGRAMS 3.0 — CACHING

A simpler example of a `Monoid` homomorphism.



A simpler example of a `Monoid` homomorphism.



DIAGRAMS — CURRENT MODEL

```
final case class Diagram private (
  primitives: Dual[List[Prim]], env: Envelope)

final case class Envelope(f: Option[V2 => Max[Double]])


def mkDiag(ps: List[Prim]): Diagram

def envelope(d: Diagram): Envelope

def beside(v: V2, d1: Diagram, d2: Diagram): Diagram


val diagramMonoid: Monoid[Diagram]

val envelopeMonoid: Monoid[Envelope]
```

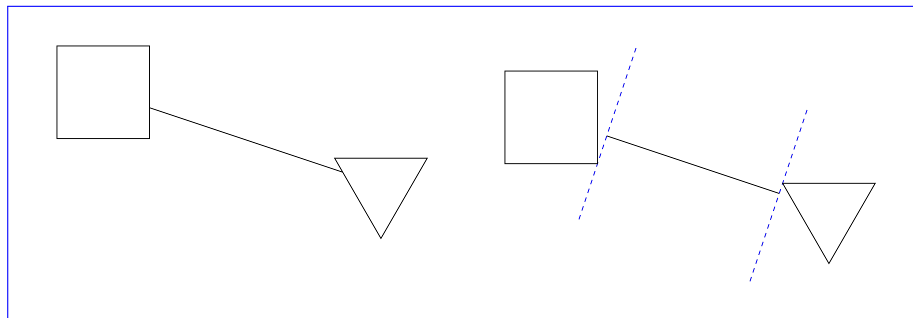
CONCLUSIONS

- In many problems composition is the most important operation.
- Try a `Monoid`/`Semigroup` as the means of composition.
- Types can declare the semantics of composition.
- Inability to define a lawful `Monoid` signals a problem in the design.
- Inability to find useful homomorphisms can signal a problem.
- Think deeply about the algebraic properties of your types.
- Read the paper!. Functional pearls make great intro papers.

Questions?

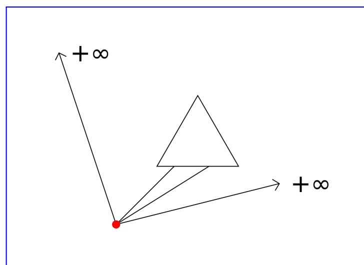
Code & slides: https://github.com/paraseba/scaladores-may-2018-talk.

How to draw a ray between diagrams?



Envelopes clearly don't work.

DIAGRAMS 4.0 — TRACES

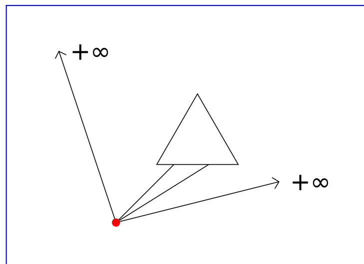


Monoids: Theme and Variations

A Trace Is (Again) a Function

```
// points on the plane
type P2 = (Double, Double)
```

DIAGRAMS 4.0 — TRACES



Monoids: Theme and Variations

A Trace Is (Again) a Function

```
// points on the plane
type P2 = (Double, Double)

final case class Trace(f: P2 => V2 => Double)
```

DIAGRAMS 4.0 — TRACES

How do Traces compose?

A Semigroup for Traces

```
// libraries provide this type out of the box
final case class Min[A](unmin: A)

implicit def minSemi[A:Ordering] = new Semigroup[Min[A]] {
  def append(a: Min[A], b: => Min[A]): Min[A] =
    Min(Ordering[A].min(a.unmin, b.unmin))
}

// final case class Trace(f: P2 => V2 => Double)
final case class Trace(f: P2 => V2 => Min[Double])

val semiGroup: Semigroup[Trace] = wSemi(_.f, Trace(_))
```

DIAGRAMS 4.0 — TRACES

But we push for a `Monoid`, not just a `Semigroup`!

Which One Should We Use?

```
// P2 => V2 => Min[Double]

Option[P2 =>        V2 => Min[Double]]
       P2 => Option[V2 => Min[Double]]
       P2 =>        V2 => Option[Min[Double]]
```

DIAGRAMS 4.0 — TRACES

The Trace Monoid

```
final case class Trace(f: P2 => V2 => Option[Min[Double]])

def traceP(p: Prim): Trace = {...}

object Trace{
  implicit val traceMonoid: Monoid[Trace] = wMon(_.f, Trace(_))

  def trace(d: Diagram): Trace =
    mconcat(d.primitives.undual.map(traceP))
}
```

Again, full expressiveness for `Trace` type!
The process is simple: find the right `Monoid`, `mconcat` all primitives.