Textanalysis with Monoids

Markus Hauck (@markus1189)

Textanalysis with Monoids

(A taste of FP)

Introduction

- IT Consultant at codecentric since >3 years
- working in **Scala** since >5 years, **Haskell** for fun
- passionate functional programmer
- Scala and Haskell Meetup in Frankfurt — Say Hi!

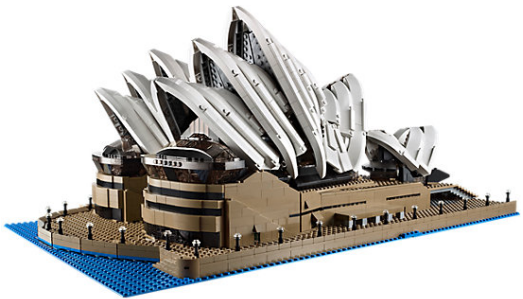

I WANT YOU
FOR FUNCTIONAL PROGRAMMING
imgflip.com

Content

- next slides: why FP?
- typeclasses (?)
- monoids
- case study: text analysis
- extensions
- conclusion

Lego vs Duplo

Lego vs Duplo



pictures from shop.lego.com

Lego vs Duplo

- **Duplo** favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse

Lego vs Duplo

- **Duplo** favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse
- **Lego** focuses on small composable building blocks
  - blocks can conveniently be reused for other purposes
  - limited use of specialized building blocks

Lego vs Duplo

- **Duplo** favours large specialized building blocks
  - blocks tend to be too big
  - limited reuse
- **Lego** focuses on small composable building blocks
  - blocks can conveniently be reused for other purposes
  - limited use of specialized building blocks

OO tends to be like **Duplo**, **FP** tends to be like **Lego**

Typeclasses

- forget the “class” part again (too overloaded)
- a way to implement overloaded functions
- not (yet) first class in scala
- encoded as class/trait with abstract methods
- use implicit resolution to define and pass around instances

Typeclasses

```scala
abstract class Showable[A] {
  def show(x: A): String
}

object Showable {
  implicit val showableInt = new Showable[Int] {
    def show(x: Int): String = x.toString
  }
}
```

Typeclasses

- disclaimer: if you want to nitpick, “Int is a <TC-Name>” is wrong
- State,Option,List is **not** a Monad
- correct: State,Option,List has a (valid) Monad instance
- is that all? No — laws

Typeclasses — Laws

- typeclasses need laws
- otherwise it is super hard to reason about code
- at least without knowing all the instances (impossible)
- that's why custom typeclasses without laws are frowned upon
- there are still reasons, but mostly: **don't unless you know why**

The Case Study

- analyze text file/stream/...
- collect multiple metrics
- using single traversal
- similar to the `wc` commandline tool

```
bash> wc moby-dick.txt
21206  208425 1193382 moby-dick.txt
```

Monoids

```
1       0  +       1  +       5
2       1  *       2  *       5
3      ""  + "Hello"  + "World"
4  List() ++ List(4) ++ List(2)
```

Monoids

- claim: most of you are using them without knowing it
- **very** common pattern
- implemented as a typeclass + instances
- provided by FP libraries (cats)

Monoids

- Quick Recap: Monoids
- binary method `combine` and nullary method `empty`

```
1  trait Monoid[A] {
2    def empty: A
3    def combine(x: A, y: A): A
4  }
5
6  // infix operator: x |+| y === combine(x, y)
```

we need to implement:

```
1 implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
2   def empty: Int = ???
3   def combine(x: Int, y: Int): Int = ???
4 }
```

- what about

```
1 implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
2   def empty: Int = 42
3   def combine(x: Int, y: Int): Int = 1337
4 }
```

- that's what laws are for
- check using ScalaCheck / Discipline / ...

Monoid Laws

```
1  empty |+|     y === y                (left identity)
2  x     |+| empty === x                (right identity)
3  (x |+| y) |+| z === x |+| (y |+| z)  (associative)
```

- associativity: it's about order of **evaluation**
- not: commutativity, where order of operands does not matter

```scala
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine(x: Int, y: Int): Int = ???
}
```

```scala
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine(x: Int, y: Int): Int = x + y
}
```

```
1 implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
2   def empty: Int = 1
3   def combine(x: Int, y: Int): Int = ???
4 }
```

```scala
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 1
  def combine(x: Int, y: Int): Int = x * y
}
```

```scala
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = Int.MinValue
  def combine(x: Int, y: Int): Int = ???
}
```

```scala
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def empty: Int = Int.MinValue
  def combine(x: Int, y: Int): Int = x.max(y)
}
```

Monoids

```
1 Monoid.empty |+|       1 |+|       5
2 Monoid.empty |+|       2 |+|       5
3 Monoid.empty |+| "Hello" |+| "World"
4 Monoid.empty |+| List(4) |+| List(2)
```

Monoid Zoo

`List[A]` is a `Monoid`

Monoid Zoo

`List[A]`	is a `Monoid`
`A => B`	if `B` is a `Monoid`

Monoid Zoo

`List[A]`	is a `Monoid`
`A => B`	if `B` is a `Monoid`
`(A,B)`	if `A` **and** `B` are `Monoids`

Monoid Zoo

`List[A]`	is a `Monoid`
`A => B`	if `B` is a `Monoid`
`(A,B)`	if `A` **and** `B` are `Monoids`
`Future[A]`	if `A` is a `Monoid`

Monoid Zoo

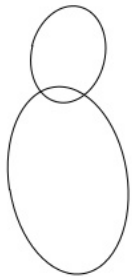
`List[A]`	is a `Monoid`
`A => B`	if `B` is a `Monoid`
`(A,B)`	if `A` **and** `B` are `Monoids`
`Future[A]`	if `A` is a `Monoid`
`Map[A,B]`	if `B` is a `Monoid`

Monoid Zoo

`List[A]`	is a `Monoid`
`A => B`	if `B` is a `Monoid`
`(A,B)`	if `A` **and** `B` are `Monoids`
`Future[A]`	if `A` is a `Monoid`
`Map[A,B]`	if `B` is a `Monoid`

```
1 val m1 = Map("as" -> 21, "bs" -> 4)
2 val m2 = Map("as" -> 21, "cs" -> 2)
3 m1 |+| m2
4 //  Map("as" -> 42, "bs" -> 4, "cs" -> 2)
```

HOW TO DRAW A DOG IN TWO EASY STEPS

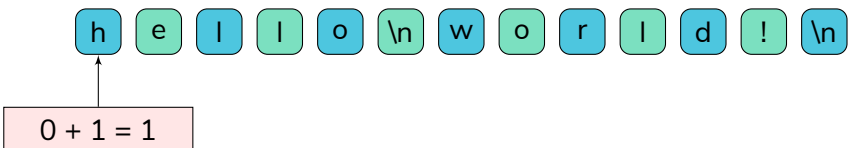


1. DRAW TWO CIRCLES, ONE FOR THE HEAD AND ONE FOR THE BODY

2. NOW DRAW THE REST OF THE DOG

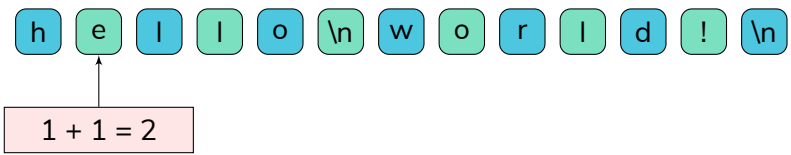
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



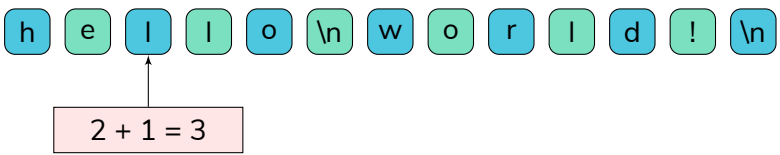
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



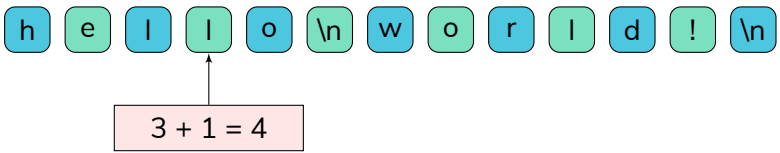
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



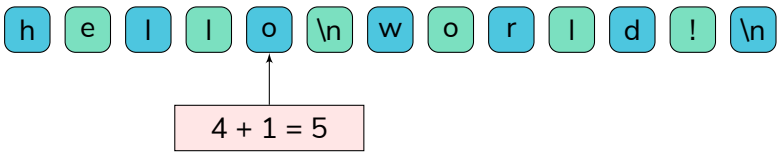
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



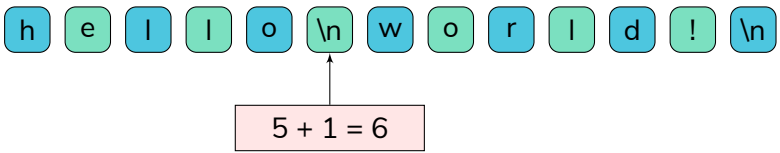
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



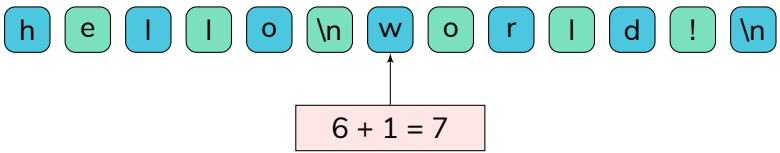
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



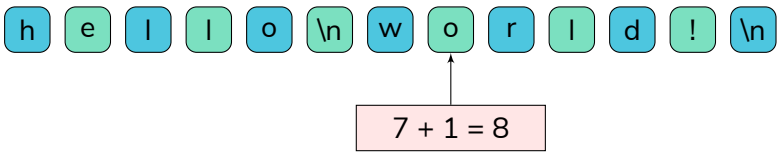
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



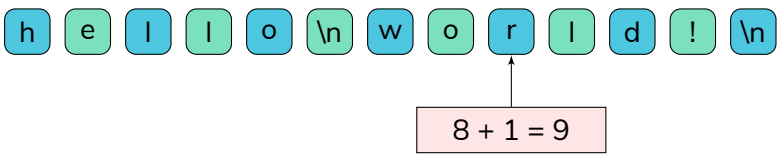
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



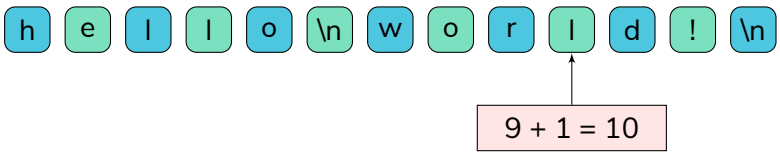
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



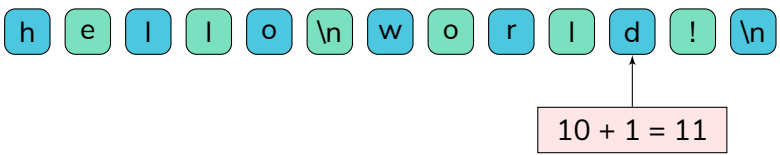
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



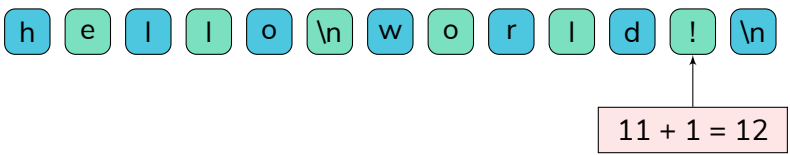
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



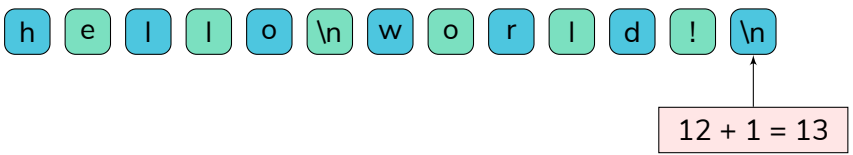
Monoids: Counting Chars

- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



Monoids: Counting Chars

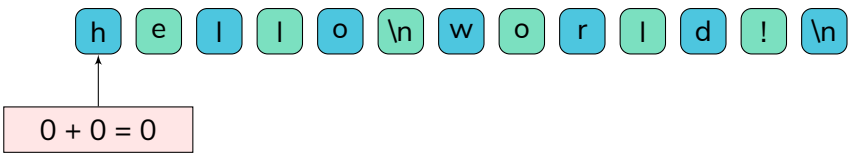
- counting chars is easy, use `(Int, +)` as a Monoid
- count 1 (`combine` 1) for every character



- so the result is 13 chars in total

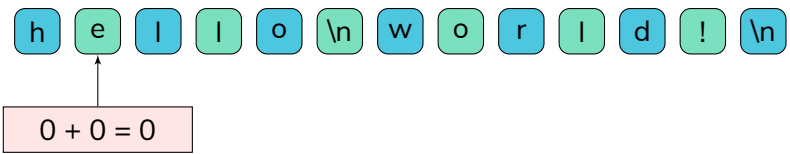
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



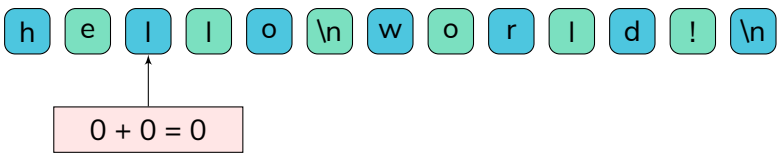
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



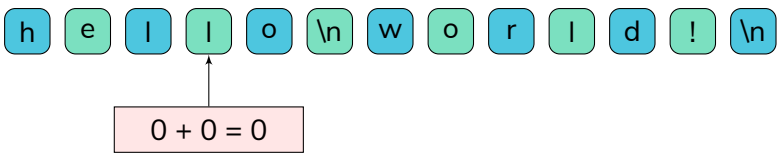
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



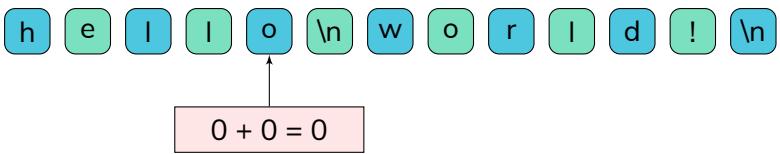
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



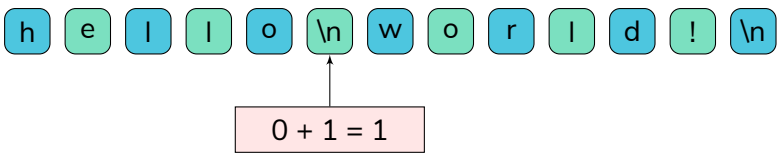
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



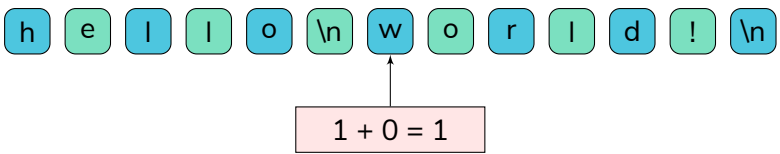
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



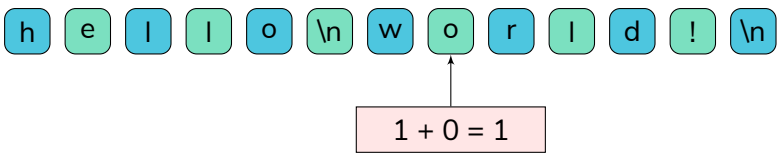
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



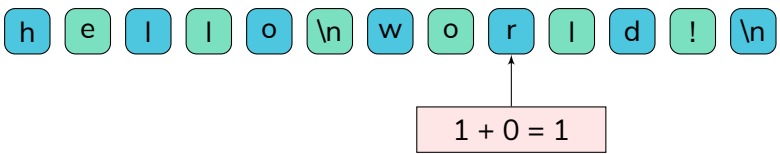
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



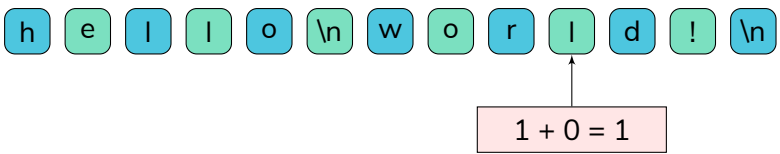
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



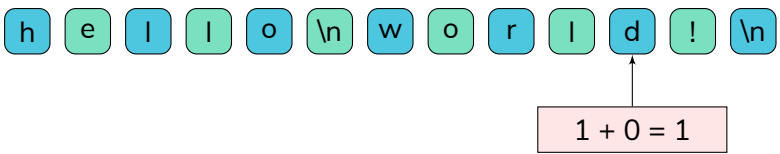
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



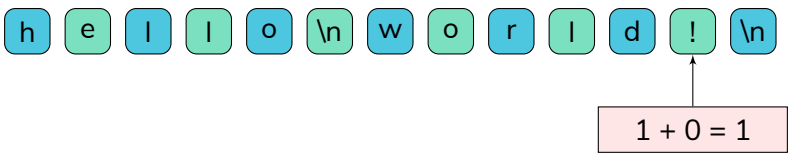
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



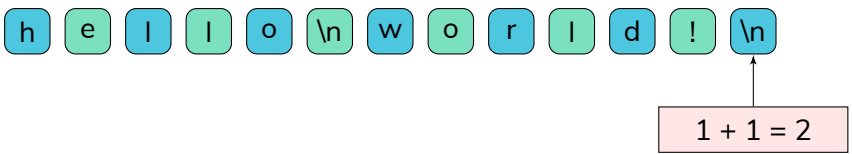
Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



Monoids: Counting Lines

- to count lines, use again `(Int, +)` as a Monoid
- but **only** count 1 if the character is a \n



- we counted 2 lines in total

Composing Monoids

- now: count chars **and** lines
- for multiple metrics, do multiple passes?!
- no — because monoids **compose**
  - inductive: monoid + base monoid
  - product: tuple of monoids

Monoid Composition — Induction

- some Monoids are based inductively on others

```
1  def optionMonoid[A: Monoid] = new Monoid[Option[A]] { /*...*/ }
```

- Option, Future, IO, Task, ...
- the Option-Monoid works like this:

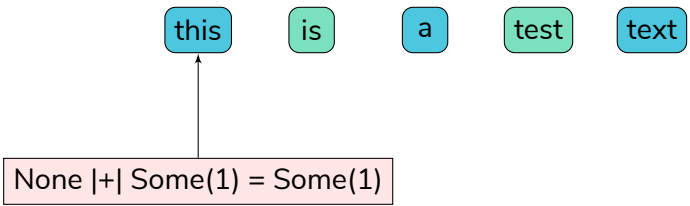
```
1  None |+| y === y
2  x |+| None === x
3  Some(x) |+| Some(y) === Some(x |+| y)
```

Monoid Composition — Option And Stopwords

- as an example: filter out (don't count) stopwords
- stopwords = most common words that are not interesting ("the", "a", ...)
- idea: if it is a stopword, use `None`, otherwise regular count with `Some`
- change of scenario: we now have an `Iterator[String]`

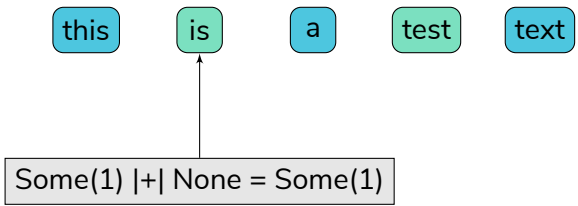
Monoid Composition — Option And Stopwords

- assuming both “is” and “a” are classified as stopwords:



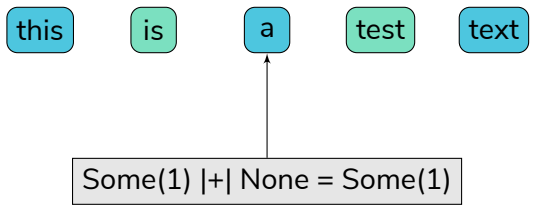
Monoid Composition — Option And Stopwords

- assuming both “is” and “a” are classified as stopwords:



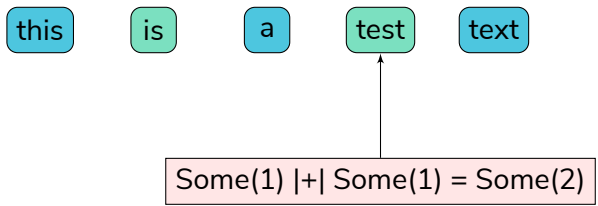
Monoid Composition — Option And Stopwords

- assuming both “is” and “a” are classified as stopwords:



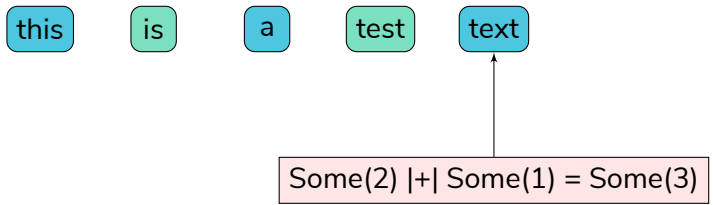
Monoid Composition — Option And Stopwords

- assuming both “is” and “a” are classified as stopwords:



Monoid Composition — Option And Stopwords

- assuming both “is” and “a” are classified as stopwords:



- count without stopwords is 3

Monoid Composition — Option And Stopwords

- use `Option` plus `Max,Min` to get longest/shortest non-stopword
- more options:
  - don't count chars like `!?,.` etc. using Option again
  - use `Future`/`Task`/`IO` to get parallelism
  - and sooo much more

Monoid Composition — Induction

- base instance does not have to be a Monoid
- using `Option` we can lift any `Semigroup`
- `empty` becomes `None`
- useful for e.g. `Max` and `Min` to represent lower/upper bound

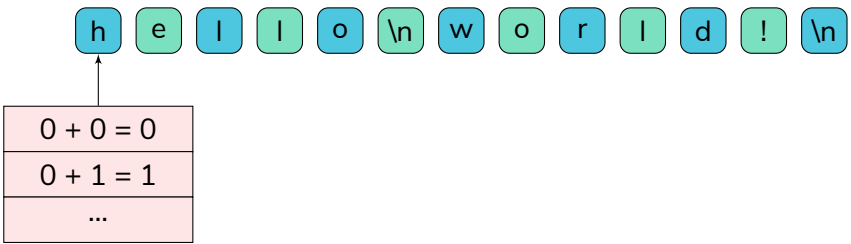
Monoid Composition — Tuple

- if `A` and `B` have a Monoid instance, so does `(A,B)`

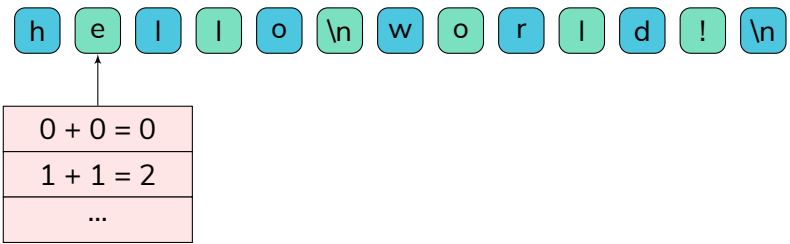
```
1  def tupleMonoid[A: Monoid, B: Monoid]: Monoid[(A, B)] =
2    new Monoid[(A, B)] {
3      def empty = (Monoid[A].empty, Monoid[B].empty)
4
5      def combine(x: (A, B), y: (A, B)) = (x._1 |+| y._1, x._2 |+| y._2)
6    }
```

- combine the two `A`'s and the two `B`'s
- we can fuse our two metrics!

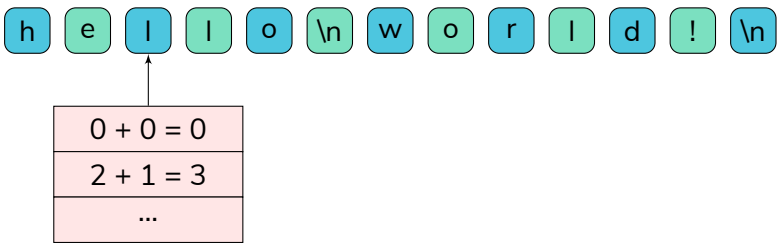
Multiple Monoids — One Traversal



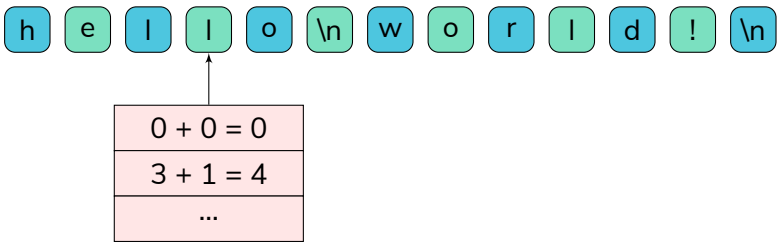
Multiple Monoids — One Traversal



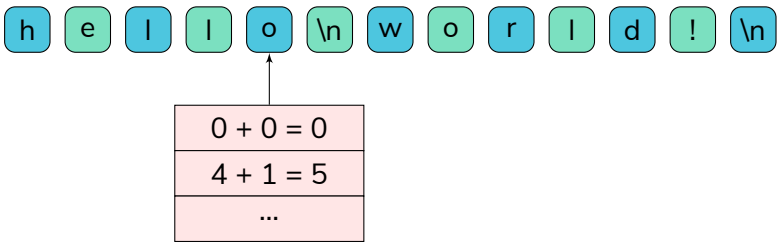
Multiple Monoids — One Traversal



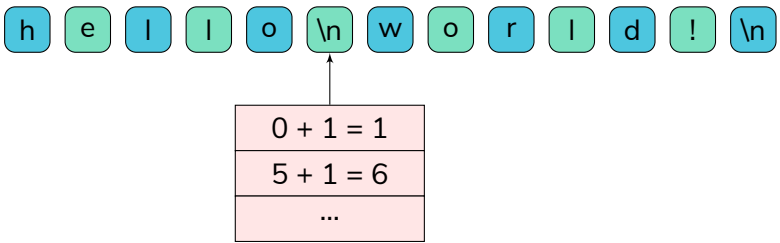
Multiple Monoids — One Traversal



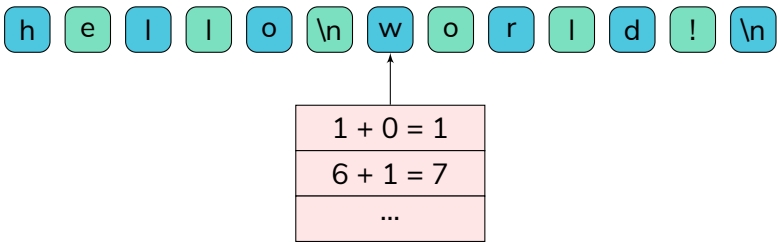
Multiple Monoids — One Traversal



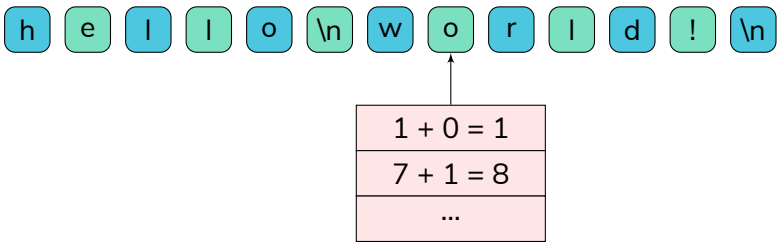
Multiple Monoids — One Traversal



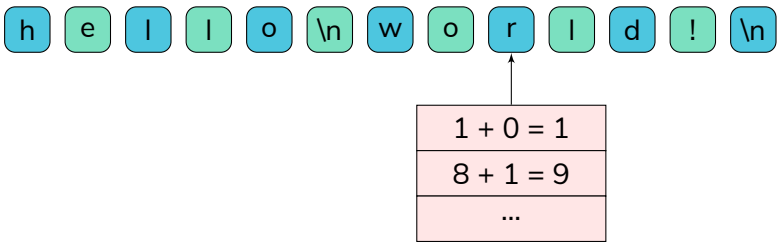
Multiple Monoids — One Traversal



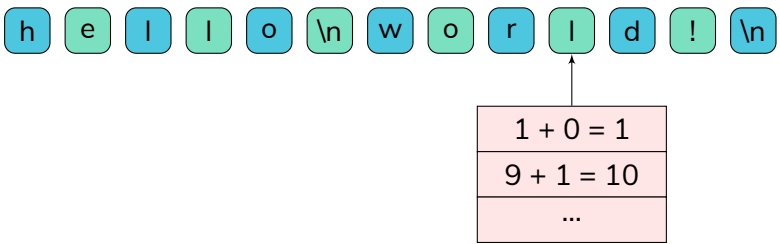
Multiple Monoids — One Traversal



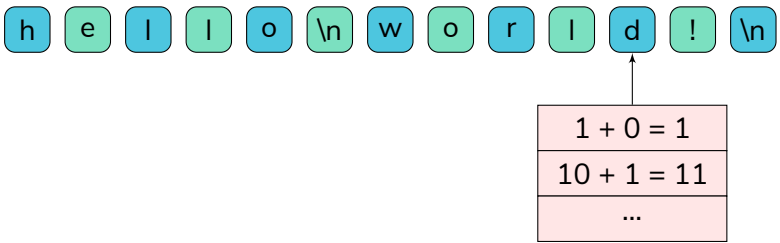
Multiple Monoids — One Traversal



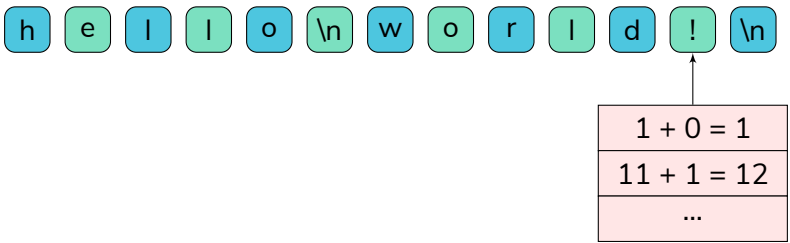
Multiple Monoids — One Traversal



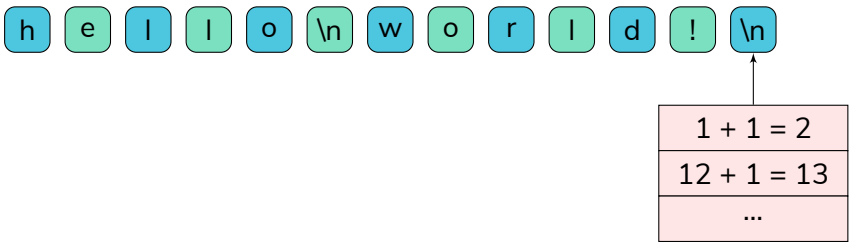
Multiple Monoids — One Traversal



Multiple Monoids — One Traversal



Multiple Monoids — One Traversal



- result: 2 lines and 13 chars

Monoids Compose (Lego Principle)

```
1  Config => A
```

Monoids Compose (Lego Principle)

```
1  Config => A
```

```
1  Config => Future[A]
```

Monoids Compose (Lego Principle)

```
1  Config => A
1  Config => Future[A]
1  Config => Future[Map[String,A]]
```

Monoids Compose (Lego Principle)

```
1  Config => A
1  Config => Future[A]
1  Config => Future[Map[String,A]]
1  Config => Future[Map[String,(A,B)]]
```

Monoids Compose (Lego Principle)

```
1  Config => A
1  Config => Future[A]
1  Config => Future[Map[String,A]]
1  Config => Future[Map[String,(A,B)]]
1  Config => Future[Map[String,(A,Option[B])]]
```

Monoids Compose (Lego Principle)

```
1 Config => A
1 Config => Future[A]
1 Config => Future[Map[String,A]]
1 Config => Future[Map[String,(A,B)]]
1 Config => Future[Map[String,(A,Option[B])]]
1 Config => Future[Map[String,(Set[A],Option[B])]]
```

- yep, still a Monoid

Extensions

From Monoids to Folds

- our framework:

```
1 def expand[A, M:Monoid](element: A): M = ??? // convert input element
2
3 val input = ??? // something that has fold
4
5 val result = input.map(expand).foldItWithMyProvidedMonoidDamnIt
```

From Monoids to Folds

- our framework:

```
1 def expand[A, M:Monoid](element: A): M = ??? // convert input element
2
3 val input = ??? // something that has fold
4
5 val result = input.map(expand).foldItWithMyProvidedMonoidDamnIt
```

- luckily, there is a typeclass over “foldable” things

Foldable

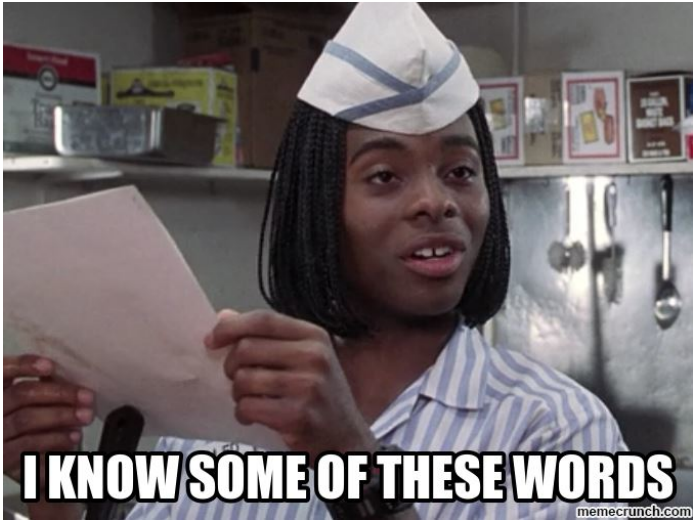
```scala
trait Foldable[F[_]:Functor] {
  // rest omitted
  // has also foldLeft, foldRight, fold, ...

  def foldMap[A, M:Monoid](fa: F[A])(f: A => M): M
}
```

- that means we are able to fold almost everything
- if there is no `Foldable`, write instance or define it yourself
- let's try Spark's `RDD`

RDDs and Folds

```
1  abstract class RDD[T] {
2    /**
3     * Aggregate the elements of each partition,
4     * and then the results for all the partitions,
5     * using a given associative function and a
6     * neutral "zero value".
7     */
8    def fold(zeroValue: T)(op: (T, T) => T): T
9  }
```

RDDs and Folds



RDDs and Folds

```
1  abstract class RDD[T] {
2    /**
3     * ..........................................
4     * ..............................................
5     * using a given associative function and a
6     * neutral "zero value".
7     */
8    def fold(zeroValue: T)(op: (T, T) => T): T
9  }
```

Monoidal RDDs

```
1  implicit class MonoidRDD[T](val rdd: RDD[T]) {
2
3    // avoid conflicts with fold/reduce etc
4    def combine(implicit M: Monoid[T]): T =
5      rdd.fold(M.empty)(M.combine(_,_))
6  }
```

The Program

```
1  def expand(w: String) = (1, w.length, Map(w -> 1))
2
3  val sc: SparkContext = ???
4  val file: String = ???
5
6  val input = sc.textFile(file).  // read the file
7    flatMap(_.split("""\W+""")). // split into words
8    map(expand)                  // action!
9
10 val (words,chars,wordMap) = data.combine
```

Streaming

- we can also integrate easily with streaming frameworks:

```
1  // Using akka-streams
2  def sinkFoldMap[A, M:Monoid](f: A => M): Sink[A, Future[M]] =
3    Sink.fold[M, A](Monoid[M].empty)((m,a) => m |+| f(a))
```

```
1  // Using fs2, already built-in
2  def foldMap[O2](f: O => O2)(implicit O2: Monoid[O2]): Stream[F, O2]
```

Conclusion

- flexible and `composable` way to cacluate metrics over text
- using `Monoid` and any `fold`, cats: `Foldable`
- easily adaptable to other frameworks like Apache Spark or fs2/Akka Streams

References

Questions?