Functional Workshop: Part 1

Boris Buliga, Valentyn Vakatsiienko

June 19, 2019

About us

Valik

Server guild manager in Kyiv. Formerly forced people to use functional programming style in the Domains (Premium) team. Now works on Tagless Infra to provide you with the best tools for your daily needs. Which are all functional, of course.

About us

Valik

Server guild manager in Kyiv. Formerly forced people to use functional programming style in the Domains (Premium) team. Now works on Tagless Infra to provide you with the best tools for your daily needs. Which are all functional, of course.

Boris

Developer at Payments by Wix team. Jumps between two extremes - Emacs Lisp and Haskell. Wants to force people around to use both languages, but can't explain why.

About the Workshop

- Basic workshop is split into 3 parts:
  1. Type classes, Semigroups and Monoids.
  2. Functors and Applicative Functors.
  3. Monads.
- Theory and practice. You'll have to write code, so make sure your laptop is ready for this.
- Main target audience is Scala developers.
- Workshop is duplicated in Haskell.

Whys

- Functional programming roams (a bit).
  - Tagless infra grows and spreads.
  - More projects are using functional programming techniques and idioms (at different scale).
- Some people are still confused by all these functional talks (`OptionT`, type lambdas etc).
- Having a common language and understanding of some fundamental stuff is important.

Plan

- Type classes
- Property based testing (just a tip of)
- Semigroups
- Monoids
- 3 interesting™ tasks

Fulfilling a dream

- Imagine that we are game developers (yay!).
- Imagine that we are writing a game (double yay!).
- And every game needs a hero, who can properly introduce itself.

Meet the hero

```scala
case class Hero(name: String, job: String, level: Int) {
  def introduce(): String = s"Hi! My name is $name. I am $level level $job."
}

object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)

  println(player.introduce())
}

// Hi! My name is Valik. I am 20 level Black Mage.
```

Every hero needs a monster

```scala
case class Hero(name: String, job: String, level: Int) {
  def introduce(): String = s"Hi! My name is $name. I am $level level $job."
}

case class Orc(name: String, level: Int) {
  def introduce(): String =
    s"Lok-tar ogar! Me be $name. Me be strong. Level $level strong!"
}

case class Ooze(level: Int) {
  def introduce(): String = 1.to(level).map(_=>"brlup").mkString("-")
}

object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)
  val orc = Orc("Garrosh", 105)
  val ooze = Ooze(2)

  println(player.introduce())
  println(orc.introduce())
  println(ooze.introduce())
}

// Hi! My name is Valik. I am 20 level Black Mage.
// Lok-tar ogar! Me be Garrosh. Me be strong. Level 105 strong!
// brlup-brlup
```

Everyone is polite

- In our world everyone is polite. Even ooze, brlup-blup!
- Rendering process of greetings is a hard process and boils down to the string manipulation.
- So we decided to avoid code repetition and write a function that does just that.

```scala
object Game extends App {
  def introduce(phrase: String): Unit = {
    // some real shit animations
    println(phrase)
    // some real shit animations
  }
  introduce(hero.introduce())
  introduce(orc.introduce())
  introduce(ooze.introduce())
}
```

- But it would so nice to avoid all this `_.introduce()`.

Introducing abstractions

```scala
trait Introducible {
  def introduce(): String
}

case class Hero(name: String, job: String, level: Int) extends Introducible {
  override def introduce(): String = s"Hi! My name is $name. I am $level level $job."
}

case class Orc(name: String, level: Int) extends Introducible {
  override def introduce(): String =
    s"Lok-tar ogar! Me be $name. Me be strong. Level $level strong!"
}

case class Ooze(level: Int) extends Introducible {
  override def introduce(): String = 1.to(level).map(_=>"brlup").mkString("-")
}
```

Using the trait

```
object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)
  val orc = Orc("Garrosh", 105)
  val ooze = Ooze(2)

  def introduce(creatute: Introducible): Unit = {
    // some real shit animations
    println(creatute.introduce())
    // some real shit animations
  }

  introduce(player)
  introduce(orc)
  introduce(ooze)
}

// Hi! My name is Valik. I am 20 level Black Mage.
// Lok-tar ogar! Me be Garrosh. Me be strong. Level 105 strong!
// brlup-brlup
```

Here comes the cockatrice

```
import io.proprietary.monsters.cockatrice._

/*...*/

object Game extends App {
  /*...*/

  val cockatrice = Cockatrice(level = 666, element = Element.Fire)

  introduce(cockatrice) // ???
                        // ain't gonna work
}
```

Shawarma to the rescue

```
import io.proprietary.monsters.cockatrice._

/*...*/

case class CockatriceWrapper(cockatrice: Cockatrice) extends Introducible {
  override def introduce(): String = {
    import cockatrice._
    s"Haha. I am a ${element.shortName} cockatrice of level ${level}."
  }
}

object Game extends App {
  /*...*/

  val cockatrice = Cockatrice(level = 666, element = Element.Fire)
  val cockatriceW = CockatriceWrapper(cockatrice)

  introduce(cockatriceW)

  /*...*/
}


// Haha. I am a fire cockatrice of level 666.
```

Properties we care about

- Abstraction - we care about what you can do and not what you are.
- Composition - we want a way to express that we want something that can do several things at once.
- Extensibility - we want to extend even types that we don't own. Built-in types as well.

With `trait` + wrappers approach

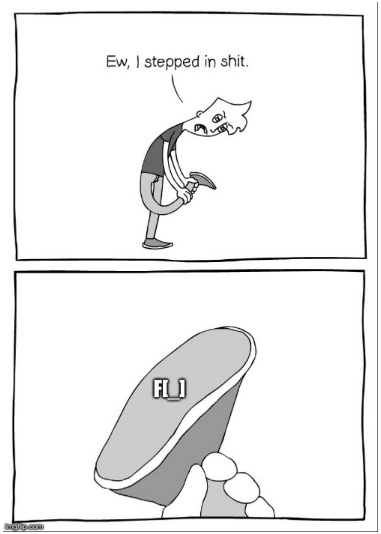
- Abstraction works. We are able to define generic `introduce` function.
- Composition works thanks to `with` keyword.
  ```
  def surpriseAttack[A <: Introducible with CanAttack](creature: A): Unit
  ```
- Extensibility is possible with some caveats:
  - No consistency - we wrap only types we don't own.
  - Nightmare to maintain when you need several behaviours. So you wrap the wrapper.
  - Bad usability - you can't interchangeably use wrapper and the underlying value.

With `trait` + wrappers approach

- Abstraction works. We are able to define generic `introduce` function.
- Composition works thanks to `with` keyword.
  ```
  def surpriseAttack[A <: Introducible with CanAttack](creature: A): Unit
  ```
- Extensibility is possible with some caveats:
  - No consistency - we wrap only types we don't own.
  - Nightmare to maintain when you need several behaviours. So you wrap the wrapper.
  - Bad usability - you can't interchangeably use wrapper and the underlying value.

You know where it's going to, right?

F[_]



What if. . .

We divide data and behaviour

What if. . .

We divide data and behaviour

```
trait Introducible[A] {
  def introduce(a: A): String
}
```

- `A` is data
- `Introducible[A]` - behaviour

What if. . .

We divide data and behaviour

```scala
trait Introducible[A] {
  def introduce(a: A): String
}
```

- `A` is data
- `Introducible[A]` - behaviour

So we can pass separately two values:

- data itself
- implementation of behaviour

What if. . .

We divide data and behaviour

```
trait Introducible[A] {
  def introduce(a: A): String
}
```

- `A` is data
- `Introducible[A]` - behaviour

So we can pass separately two values:

- data itself
- implementation of behaviour

```
def introduce[A](creature: A)(impl: Introducible[A]): Unit =
  println(impl.introduce(creature))
```

We treat our own types

```scala
trait Introducible[A] {
  def introduce(a: A): String
}

case class Hero(name: String, job: String, level: Int)

object Hero {
  val introducibleHero: Introducible[Hero] = (a: Hero) => {
    import a._
    s"Hi! My name is $name. I am $level level $job."
  }
}

object Game extends App {
  def introduce[A](creature: A)(ev: Introducible[A]): Unit =
    println(ev.introduce(creature))

  introduce(Hero(name = "Valik", job = "Black Mage", level = 20))(Hero.introducibleHero)
}
```

And external types in the same way

```scala
import io.proprietary.monsters.cockatrice._

trait Introducible[A] {
  def introduce(a: A): String
}

object Introducible {
  val introducibleCockatrice: Introducible[Cockatrice] = (a: Cockatrice) => {
    s"Haha. I am a ${a.element.shortName} cockatrice of level ${a.level}."
  }
}

object Game extends App {
  def introduce[A](creature: A)(ev: Introducible[A]): Unit = println(ev.introduce(creature))

  introduce(Cockatrice(level = 666, element = Element.Fire))(Introducible.introducibleCockatrice)
}
```

But passing implementation around is. . .

Cucumbersome

So implicits :(

```scala
object Introducible {
  implicit val introducibleCockatrice: Introducible[Cockatrice] = /*...*/
}

object Hero {
  implicit val introducibleHero: Introducible[Hero] = /*...*/
}

object Game extends App {
  def introduce[A](creature: A)(implicit ev: Introducible[A]): Unit =
    println(ev.introduce(creature))

  introduce(Hero(name = "Valik", job = "Black Mage", level = 20))
  introduce(Cockatrice(level = 666, element = Element.Fire))
}
```

Summoning the summoner

```scala
trait Introducible[A] { /*...*/ }

object Introducible {
  def apply[A: Introducible]: Introducible[A] = implicitly[Introducible[A]]

  /*...*/
}

/*...*/

object GameSummoner extends App {
  def introduce[A: Introducible](creature: A): Unit =
    println(Introducible[A].introduce(creature))

  introduce(Hero(name = "Valik", job = "Black Mage", level = 20))
  introduce(Cockatrice(level = 666, element = Element.Fire))
}
```

So with type classes

- Abstraction works. Again, we are able to define generic `introduce` function.
- Composition works, just require two behaviours.

```
object Game {
  def surpriseAttack[A : Introducible : CanAttack](creatute: A): Unit
}
```

- Extensibility achieved and maintained consistency and usability.
  - Consistent - we treat our own types the same way we types we don't own.
  - Usability - no wrappers, no interchangeably problem. You pass data and behaviour implementation separately.

So type classes

Type class is just a construct that supports ad hoc polymorphism. E.g. allows one to define polymorphic functions that can be applied to arguments of different types and behave differently based the type of the arguments.
In other words, function overloading.
In Scala this can be achieved in several ways:

- Class inheritance.
- Traits.
- Type classes (traits + implicits).

Semigroup

A semigroup is an algebraic structure consisting of a set together with an associative binary operation.

Semigroup

A semigroup is an algebraic structure consisting of a set together with an associative binary operation.
A semigroup is a set S with binary operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Semigroup

A semigroup is an algebraic structure consisting of a set together with an associative binary operation.
A semigroup is a set S with binary operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Important. Semigroup is a pair of the set and the operation. You can't say that string is a semigroup, you must provide an operation. And in many cases there is more than one operation for a set to form a semigroup.

But it's not that scary

```scala
package object typeclass {

  //
  // Laws:
  //   1. ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
  //
  trait Semigroup[A] {
    def append(x: A, y: A): A
  }

  object Semigroup {
    def apply[A: Semigroup]: Semigroup[A] =
      implicitly[Semigroup[A]]
  }

}
```

Laws are important

- In most cases type classes should have some associated laws.
- Laws describe behaviour of the interface, what you can expect.
- This gives you confidence when you combine pieces of code.

Instance example

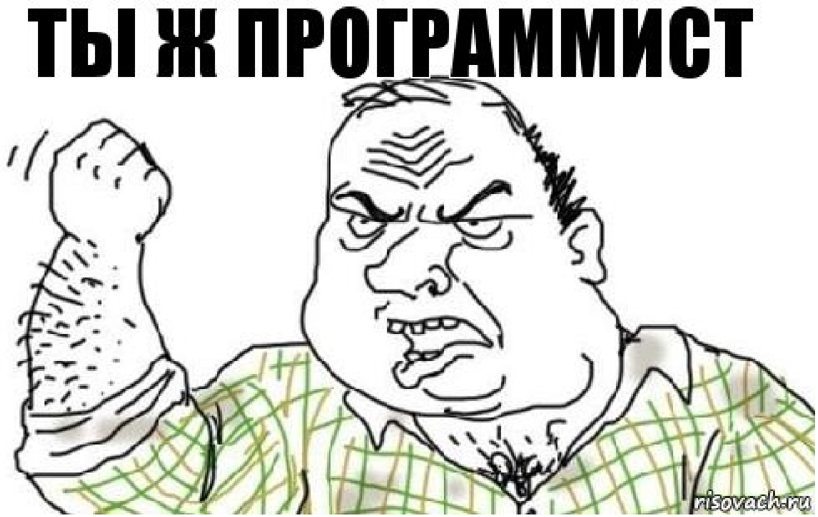
```scala
package object implicits {
  implicit val stringSemigroup: Semigroup[String] = new Semigroup[String] {
    override def combine(x: String, y: String): String = x + y
  }
}
```

Checking laws - ~~pen and paper~~ in comments

```
package object implicits {
  implicit val stringSemigroup: Semigroup[String] = new Semigroup[String] {
    override def combine(x: String, y: String): String = x + y
  }
}

/*
combine(a, combine(b, c))
  = combine(a, b + c)
  = a + (b + c)
  = (associativity of +)
  = (a + b) + c = combine(a + b, c)
  = combine(combine(a, b), c)
*/
```

You're programmer after all



Question on the interview: property based testing

```
object SemigroupSpecification extends Properties("Semigroup") with SemigroupSpecificationSupport {
 include(semigroup[String](stringSemigroup))
}

trait SemigroupSpecificationSupport {
 def semigroup[A](sg: Semigroup[A])(implicit ar: Arbitrary[A], tag: ClassTag[A]): Properties =
   new Properties(s"Semigroup[${tag.toString}]") {
     // ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
     property("associativity") = forAll { (a: A, b: A, c: A) =>
       sg.combine(sg.combine(a, b), c) =? sg.combine(a, sg.combine(b, c))
     }
   }
}

/*
+ Semigroup.Semigroup[java.lang.String].associativity: OK, passed 100 tests
  .
*/
```

More examples

- Numbers with $+$, $*$, *min*, *max*
- Booleans with conjunction, disjunction, implication etc.
- Square nonnegative matrices with multiplication.
- Lists, Strings, Maps etc. with concatenation/union

Lets hack

1. Clone `git@github.com:d12frosted/wax.git`
2. Add missing definitions in `wax.typeclass.semigroup`
3. Run tests in `wax.typeclass.semigroup`

Monoid

A monoid is an algebraic structure consisting of a set together with an associative binary operation and an identity element from that set.

Monoid

A monoid is an algebraic structure consisting of a set together with an associative binary operation and an identity element from that set.
A monoid is a set S with binary operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

and identity element e that satisfies

$$\forall a \in S : e \cdot a = a \cdot e = a$$

Monoid

A monoid is an algebraic structure consisting of a set together with an associative binary operation and an identity element from that set.
A monoid is a set S with binary operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

and identity element e that satisfies

$$\forall a \in S : e \cdot a = a \cdot e = a$$

In other words, monoid is just a semigroup with identity element.

Again, it's not that scary

```scala
package object typeclass {

  //
  // Laws:
  //   1. ∀a, b, c ∈ S : (a · b) · c = a · (b · c)
  //   2. ∀a ∈ S : e · a = a · e = a
  //
  trait Monoid[A] extends Semigroup[A] {
    def empty: A
  }

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

}
```

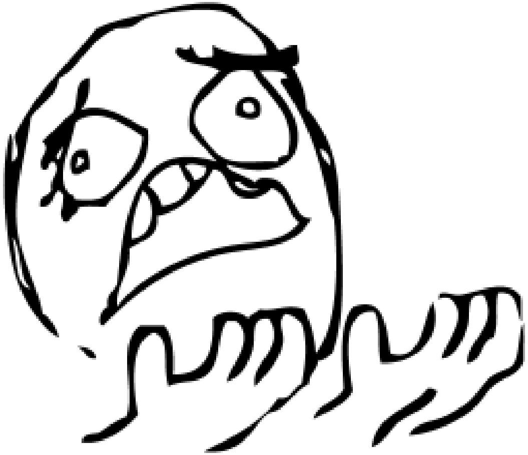
Examples

- $\{\mathbb{N}_0, +\}$, where 0 is the identity element.
- $\{\mathbb{N}, *\}$, where 1 is the identity element.
- Boolean with XOR, XNOR, OR, AND.
- String with concatenation (empty string is identity element).

But not every Semigroup forms a Monoid (we are not talking about free monoids here):

- `BigNumber` practically doesn't have identity element for `min`.

The most important question



Why did we learn this?

The Fibonacci numbers

On the interview we ask people to write a function that returns the nth Fibonacci number.

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}, \forall n > 1
\end{aligned}
$$

Solution

What we expect

```
def fib(n: Int): Int = {
  def fibTail(n: Int, a: Int, b: Int): Int = n match {
    case 0 => a
    case _ => fibTail(n - 1, b, a + b)
  }

  fibTail(n, 0, 1)
}
```

Ideal solution

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{2\phi - 1}$$

$$\phi = \frac{1 + \sqrt{5}}{2}$$

Solution

What we expect

```
def fib(n: Int): Int = {
  def fibTail(n: Int, a: Int, b: Int): Int = n match {
    case 0 => a
    case _ => fibTail(n - 1, b, a + b)
  }

  fibTail(n, 0, 1)
}
```

Ideal solution

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{2\phi - 1}$$

$$\phi = \frac{1 + \sqrt{5}}{2}$$

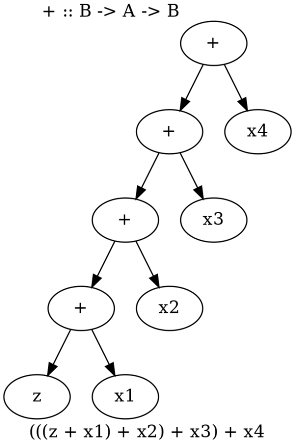
As they say, truth is somewhere in the logarithm.

Two folds

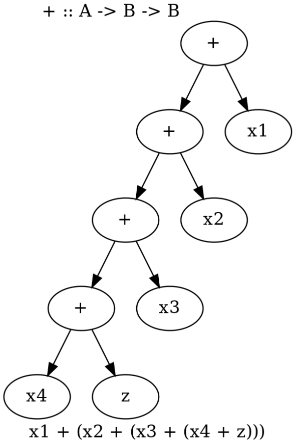
- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
  - Folds the structure from left to right
- `def foldr[A, B](xs: Seq[A])(z: B)(op: A => B => B): B`
  - Folds the structure from right to left
- Since combining function is asymmetrical in its types:
  - It's impossible to place parentheses in the arbitrary fashion or even just change the direction of the `fold`
  - It's impossible to implement a total `fold` without default value of type `B`

Example

foldl



foldr



What Monoid gives us

- Combining function is symmetrical (`append : A -> A -> A`).
- Monoid provides identity element of type `A` (`empty`).
- So we can define a special `fold`
  - `def foldMonoid[A: Monoid](xs: Seq[A]): A`
- Associativity law says that we can put parentheses in an arbitrary fashion.
- Identity law says that we can place identity element anywhere - on the far left, on the far right or even in the middle.
- Laws that we checked are giving us means to abstract implementation.
- When we ask consumer of our API to provide us a Monoid we don't want only a behaviour but behaviour that follows the laws. We want these properties as much as we want the functions.

Power in terms of Monoid

- In some cases all elements of the list are the same.

Power in terms of Monoid

- In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a)\ldots) = a^n$$

Power in terms of Monoid

- In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a) \ldots) = a^n$$

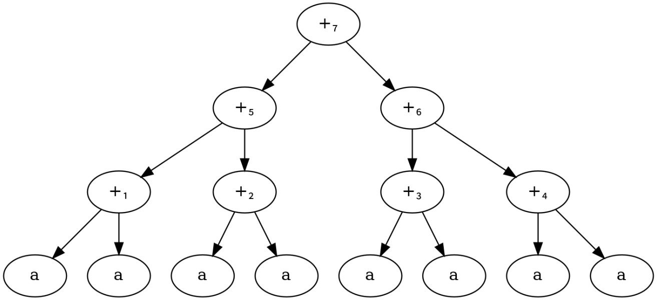
- Since we can reorder the parentheses, we can arrange them like this.

Power in terms of Monoid

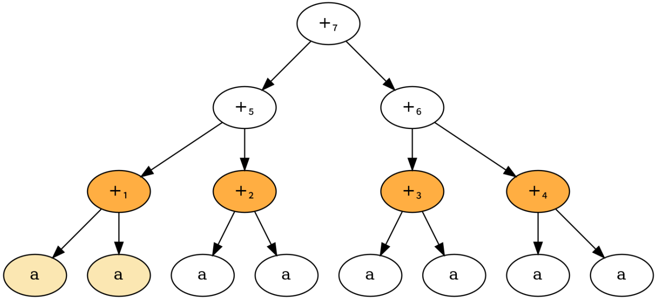
- In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a)\ldots) = a^n$$

- Since we can reorder the parentheses, we can arrange them like this.

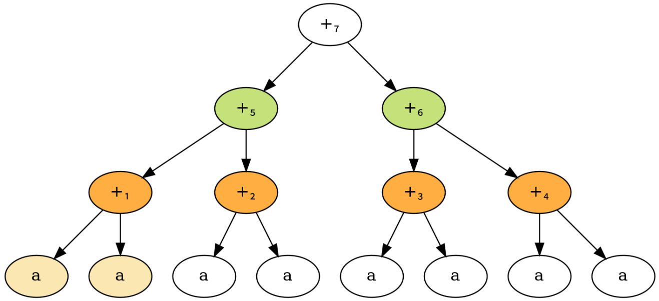


Power in terms of Monoid



Evaluating $a + a$ always yields the same result. So there is no point in repeating this calculation 4 times.

Power in terms of Monoid



The same thing with the upper level. In this particular example, we can avoid 4 operations out of 7. In general, this optimisation leads to the result in log n operations.

Power in terms of Monoid

All this means that we can define a function power:

```
def power[A: Monoid](a: A, n: Int): A = {
  ???
}
```

Back to Fibonacci

Fibonacci number can be defined in a different way.

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Back to Fibonacci

Fibonacci number can be defined in a different way.

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

- The Fibonacci number can be calculated using square nonnegative matrix multiplication.
- Square nonnegative matrices form Monoid with multiplication.
- So we can put parentheses in a way we like it.

Time for work

- Open `wax.exercise.fibonacci` module.
- Task is to implement monoid for `Matrix2x2`.
- Run `MatrixMonoidSpecification` to test your instance.
- Run `Main` object and compare benchmarking results of tail recursive and matrix-based implementations.

Outcome

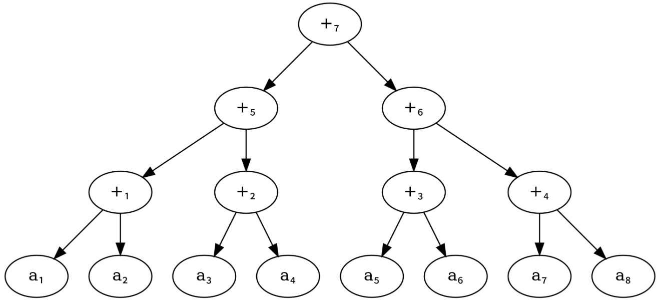
- Just think about it.
    - Giving any monoid we have a helper that allows us to efficiently calculate a^n.
    - This is not only because of the operations, but the laws (or so-called properties) that come with these operations.
    - Monoids are everywhere around us. We deal with them every day, without even noticing it.
- You might forget how matrix multiplication works, but now you remember, right?

Folds with Monoids

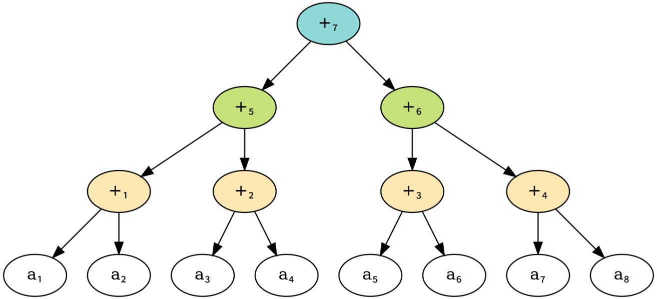
- We already know that Monoids give us an ability to place parentheses in any fashion.
- We already saw that when it comes to folding the list of the same elements we gain performance.
- But what if the elements are not equal? Do we gain anything?

Folds with Monoids

- We already know that Monoids give us an ability to place parentheses in any fashion.
- We already saw that when it comes to folding the list of the same elements we gain performance.
- But what if the elements are not equal? Do we gain anything?



Folds with Monoids



Every expression on each level does not depend on other expressions from the same level, which means that we can evaluate them in parallel.

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.
- But we can transform (going ahead, `map`) them into something that is a Monoid
    - Again, going ahead, this also can be done in parallel.
- There is a strange accent, where people pronounce 'fold' as 'reduce'.
- This is how we get the `mapReduce`.

Getting the top used words from set of books

- Open `wax.exercise.mapreduce` module.
- Task is to
  - Implement monoid for `MapReduce.Result[A]`.
  - Implement the `job` function. Let's find the most used word among all the books that is also longer than 4 symbols.
- Books are located in the `resources` directory.
- Use `allBooks` to load all available books or `authorBooks` to load all books of specific author.
  - `authorBooks("boris")` - you can use this author with small amount of text to test your `job`.
- Compare benchmark results.

Things to note

- Functional programming is not about `Monads` and `IO`.
    - Funny enough, first versions of Haskell were naked and no one dared to tell the committee that `IO` is missing.
- Functions matter.

Things to note

- Functional programming is not about `Monads` and `IO`.
  - Funny enough, first versions of Haskell were naked and no one dared to tell the committee that `IO` is missing.
- Functions matter.
- Can a function be monoid?

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`
- Can it be a monoid?
- Well, generally speaking, not! Because we know nothing about the type `A`.
- But what if `A` is a monoid?

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value:  A)`
- Can it be a monoid?
- Well, generally speaking, not! Because we know nothing about the type `A`.
- But what if `A` is a monoid?

```scala
case class Wrapper[A](value: A)

object Wrapper {
  implicit def wrapperMonoid[A: Monoid]: Monoid[Wrapper[A]] = new Monoid[Wrapper[A]] {
    override def empty: Wrapper[A] = Wrapper(Monoid[A].empty)

    override def combine(x: Wrapper[A], y: Wrapper[A]): Wrapper[A] =
      Wrapper(Monoid[A].combine(x.value, y.value))
  }
}
```

Wrappers of monoids are monoids

- So IO can also be a monoid
```
def ioMonoid[A: Monoid]: Monoid[IO[A]] = ???
```
- Which means that we can combine IO actions (in some new sense).
- Functions are wrappers (in some sense), so they also can be monoids
```
def functionMonoid[A, B: Monoid]: Monoid[Function[A, B]] = ???
```
- Which means that we can combine functions (in some new sense).

Logger

- Logger is basically a function from `String` to `IO[Unit]`.
  ```
  type Logger = String => IO[Unit]
  ```
- `Unit` forms a monoid.
- So `IO[Unit]` forms a monoid.
- So `String => IO[Unit]` forms a monoid.
- So `Logger` forms a monoid.
- So we can combine loggers
  - `combine(fileLogger, consoleLogger)` - logs both into file and to console

Logger

- Open `wax.exercise.logging` module
- Task is to implement monoid for `IO[Logger]`
- Have fun!

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.
- Associativity is a powerful property giving us an ability to solve some tasks.
    - a^n in $\log n$
    - `mapReduce`
- Monoids are everywhere. They act like a plague, once something forms a monoid, something else also begins to form a monoid.
- We want some rest after a long session of workshop.

Questions?

$\epsilon\rho\omega\tau\eta\sigma\eta$?

Thank you very much!

We hope you enjoyed this session.