@AndrzejWasowski@scholar.social
Andrzej Wąsowski
Florian Biermann

Advanced
Programming

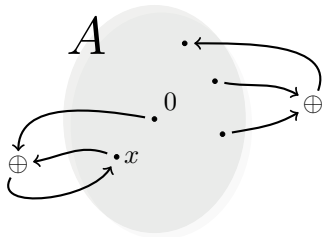## Functional Design: Monoid, Foldable, Functor, Monad

IT UNIVERSITY OF COPENHAGEN

SOFTWARE
QUALITY
RESEARCH

Monoid

Definition

- Type A
- Binary operator $\oplus : A \times A \mapsto A$ (combine)
- Zero element $0 \in A$ (empty)
- Laws ...

Associativity $\quad (x \oplus y) \oplus z \;=\; x \oplus (y \oplus z)$ for any $x, y, z \in A$

Identity $\qquad\qquad x \oplus 0 \;=\; x \;=\; 0 \oplus x$ for any $x \in A$

Associativity: `forAll { (x:A, y:A, z:A) =>`
`                    combine(combine(x, y), z) ==combine(x, combine(y, z)) }`

Identity: `forAll { (x: A) =>combine(x, empty) ==x }`
`        forAll { (x: A) =>combine(empty, x) ==x }`

Examples: (Integers, 0, +), (Integers, 1, *), (Strings, "", +)

Monoid

A type class

```scala
1 trait Monoid[A]:
2   self =>
3   def combine(a1: A, a2: A): A
4   def empty: A

5   object laws:
6     def associative (using Arbitrary[A], Equality[A]) =
7       forAll { (a1: A, a2: A, a3: A) =>
8         self.combine(self.combine(a1, a2), a3) ===
9           self.combine(a1, self.combine(a2, a3)) }

10    def unit (using Arbitrary[A], Equality[A]) =
11      forAll { (a: A) =>
12        (self.combine(a, self.empty) === a) &&
13        (self.combine(self.empty, a) === a) }

14    def monoid (using Arbitrary[A], using Equality[A]) =
15      associative && unit
```

- Type class: Monoid, type A, op, zero, abstract trait
- Type constructor: Monoid
- Laws: associativity, unit, both (imperative issues)
- Inner object's access to enclosing object
- Type constraints with implicits: Arbitrary, Equality

More on the Scalactic equality test **===** used above: https://www.scalactic.org/

Monoid

Instances

```scala
1 val stringMonoid: Monoid[String] = new:
2   def combine(a1: String, a2: String): String =
3     a1 + a2
4   val empty: String = ""

5 ...

7 object MonoidSpec extends ...
8   property("stringMonoid is a monoid") =
9     stringMonoid.laws.monoid
```

- A type class instance shows evidence that we can see a type as a Monoid
- Note that a type class instance is a value! How do we create one? We instantiate the trait.
- In place instantiation of abstract traits
- How do we "prove" that an instance is valid? We property test the Monoid Laws!
- Exercise: Take a piece of paper to write an instance for Int, 1, and multiplication
- Mentimeter on more instances

Monoid

Morphisms

- Monoid $(M, \oplus, 0)$, monoid $(N, \otimes, 1)$
- **Homomorphism** $\equiv f : M \to N$ + laws below
- Homomorphisms both ways $\equiv$ **isomorphism**
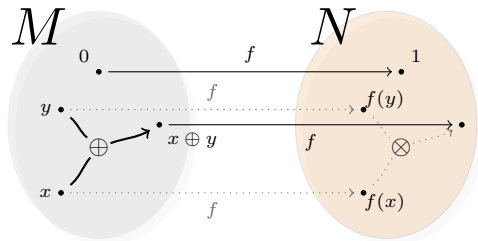
Distributive $\qquad f(x \oplus y) \; = \; f(x) \otimes f(y) \quad$ for any $x, y \in M$
Preserves identity $\quad f(0) \; = \; 1$

Distributive: `forAll { (x:A, y: A) =>f(M.combine(x, y)) ===N.combine(f(x), f(y)) }`
Preserves identity: `f(M.empty) ===N.empty`

Example homomorphism: `f(s: String): Int =s.size`
from Strings (with "" and concat) to integers (with 0 and +)
e.g. `"foo".size + "bar".size ==("foo"+ "bar").size`

Example isomorphisms: `String` and `List[Char]` through `toList` and `mkString`,
(`Boolean`, `false`, `||`) and (`Boolean`, `true`, `&&`) through negation

Foldable

Type Class

```scala
1 trait Foldable[F[_]]:
2   extension [A](as: F[A])
3     // map into a monoid and reduce using 'combine'
4     def foldMap[B: Mondoid](f: A => B): B

6     def foldRight[B](z: B)(f: (A, B) => B): B
7     def foldLeft[B] (z: B)(f: (B, A) => B): B

9     // If A is a monoid then
10    def concatenate[A](as: F[A])(using m: Monoid[A]): A =
11      as.foldLeft(m.empty)(m.combine)
```

Examples: List, IndexedSeq, Stream, Option (almost all structures implemented in the course so far!) Also trees are foldable, any iterator can be cast as foldable

- Understand the type class and the interface
- Foldable is a **higher kind**, parameterized with a type constructor
- So type classes exist both for types and for higher kinds
- Folds work with Monoids operator, no diff btw left and right. **Why?**
- Reduce = a fold from monoid's empty with combine
- MapReduce (spark) can distribute calculations in monoids
- Behavioral interface, like Enumerable, Iterable, etc. – (Monoid could be called "Addable")
- Mentimeter ($\times 2$)

Functor

Examples

Consider the following map functions from different parts of the course:

```
def map[A,B] (ga: Gen[A]) (f: A =>B): Gen[B]
def map[A,B] (la: List[A]) (f: A =>B): List[B]
def map[A,B] (oa: Option[A]) (f: A =>B): Option[B]
```

- All very similar, also `State`, `Par`, `Parser`, `Tree`
- Follow the same interface
- We could call it `Mappable`
- But for the sake of tradition we settle on `Functor`

Functor

Type Class

```scala
1 trait Functor[F[_]]:

3   extension [A](fa: F[A])
4     def map[B](f: A => B): F[B]

6   extension [A, B](fab: F[(A, B)])
7     def distribute: (F[A], F[B])=
8       (fab.map { _._1 }, fab.map { _._2 })

10  object functorLaws:
11    def map[A](using Arbitrary[F[A]], Equality[F[A]]): Prop =
12      forAll { (fa: F[A]) => fa.map[A](identity[A]) === fa }
```

Examples: List, Gen, Option, Stream are all functors (they have map, mappables)

Functor

Distribute


```
: F[(Name, Age)]                        : (F[Name], F[Age])
  ┌──────────────┐                        ┌─────────┐ ┌─────┐
  │ (John, 35)   │                        │  John   │ │ 35  │
  ├──────────────┤                        ├─────────┤ ├─────┤
  │ (Andrzej, 39)│     distribute         │ Andrzej │ │ 39  │
  ├──────────────┤   ───────────→         ├─────────┤,├─────┤
  │ (Stefan, 18) │                        │ Stefan  │ │ 18  │
  ├──────────────┤                        ├─────────┤ ├─────┤
  │     ...      │                        │   ...   │ │ ... │
  └──────────────┘                        └─────────┘ └─────┘
```

```
extension [A, B](fab: F[(A, B)])
  def distribute: (F[A], F[B])=
    (fab.map { _._1 }, fab.map { _._2 })
```

Note: This works for List and Stream but also for Gen and Par with the same single implementation! This will continue working for any type constructor in the future for which we add a Functor instance.

**Exercise:** Doodle an instance for Option (go back to the previous slide)

Map2

Some of the map2 functions from different parts of the course

```scala
def map2[A,B,C] (fa: Option[A], fb: Option[B]) (f: (A,B) => C): Option[C] =
  fa.flatMap { a => fb.map  { b => f (a,b) } }

def map2[A,B,C] (fa: List[A], fb: List[B]) (f: (A,B) => C): List[C] =
  fa.flatMap { a => fb.map  { b => f (a,b) } }

def map2[A,B,C] (fa: Gen[A], fb: Gen[B]) (f: (A,B) => C): Gen[C] =
  fa.flatMap { a => fb.map  { b => f (a,b) } }
```

- All very similar, also `State`, `Par`, `Parser`, the Map2-appable trait/interface (ApplicativeFunctor)
- `map2` is a 'kind-of' generalized binary zipWith (Beware! A different intuition for each instance!)
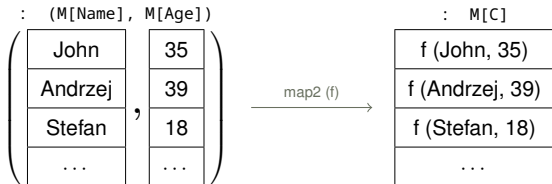
- Pic makes "sense" for `Gen`
- What happens for `List` ?
- What happens for `Option` ?
- `Parser` ?

Monad

Type Class

```scala
1 trait Monad[F[_]]
2   extends Functor[F]:
3
4   def unit[A](a: => A): F[A]
5
6   extension [A](fa: F[A])
7     def flatMap[B](f: A => F[B]): F[B]
8
9     def map[B](f: A => B): F[B] =
10       fa.flatMap[B] { a => unit(f(a)) }
```

- Understand the type class and the interface
- Monad is a higher kind, parameterized with a type constructor
- Monad is both a functor and an applicative functor
- It could be called "FlatMappable"
- It captures computations that can be sequenced, and transformed

Examples: All types in ADPRO!

Monads in the Scala standard library [21]: FilterMonadic, Stream, StreamWithFilter, TraversableMethods, Iterator, ParIterableLike, ParIterableLike, ParIterableViewLike, TraversableLike, WithFilter, MonadOps, TraversableProxyLike, TraversableViewLike, LeftProjection, RightProjection, Option, WithFilter, Responder, Zipped, ControlContext, Parser

Monad Laws

```scala
1   object monadLaws:

3     def associative[A, B, C](using Arbitrary[F[A]],
4       Arbitrary[A => F[B]], Arbitrary[B => F[C]], Equality[F[C]]): Prop =
5       forAll { (x: F[A], f: A => F[B], g: B => F[C]) =>
6         ( x.flatMap[B](f).flatMap[C](g) ) === ( x.flatMap[C] { a => f(a).flatMap[C](g) } )
7       }

9     def identityRight[A](using Arbitrary[F[A]], Arbitrary[A => F[A]], Equality[F[A]]) =
10      forAll { (x: F[A], f: A => F[A]) => x.flatMap[A](unit[A]) === x }

12    def identityLeft[A: Arbitrary](using Arbitrary[A => F[A]], Equality[F[A]]) =
13      forAll { (y: A, f: A => F[A]) => unit[A](y).flatMap[A](f)  === f(y) }
```
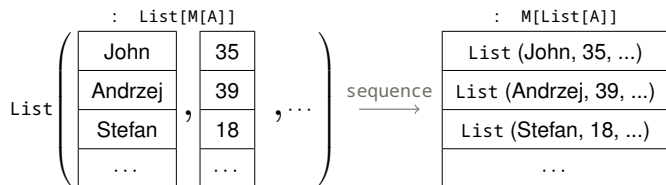
Sequence is generalized n-ary zip

```
def sequence[A] (lfa: List[F[A]]): F[List[A]]
```

- We can derive sequence for any `Monad` instance.
- The function is implemented in terms of the above interface
- The function produces a computation that produces lists. The computed list is build by polling each of the computations in the parameter for their head, and simply recomposing to the list.

- Makes "sense" for `Gen`
- What happens for `Option` ?
- What happens for `List` ?
- Parser is a bit similar to `Option` and `Gen`