@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Functional Design: Monoid, Foldable, Functor, Monad

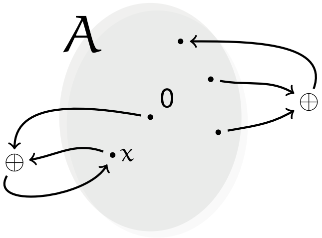
IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Monoid

Definition

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



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

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

Associativity: `forAll { (x:A,y:A,z:A) => op (op (x,y), z) should be (op (x, op (y,z))) }`

Identity: `forAll { x: A => op (x, zero) should be (x) }`

`forAll { x: A => op (zero, x) should be (x) }`

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

Monoid

A type class, an interface, a trait

- 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
- Note: Type classes (Equality) are used in Scalatest, despite of imperative style

```
trait Monoid[A] { self =>
  def op (a1: A, a2: A): A
  def zero: A

  object Laws {
    def associative (implicit arbA: Arbitrary[A], eqA: Equality[A]) =
      forAll { (a1: A, a2: A, a3: A) =>
        self.op (self.op (a1, a2), a3) should
          === { self.op (a1, self.op (a2, a3)) } }

    def unit (implicit arbA: Arbitrary[A], eqA: Equality[A]) =
      forAll { a: A =>
        self.op (a, self.zero) should === (a)
        self.op (self.zero, a) should === (a) }

    def monoid (implicit arbA: Arbitrary[A], eqA: Equality[A]) =
      { associative; unit }
```

Monoid

Instances

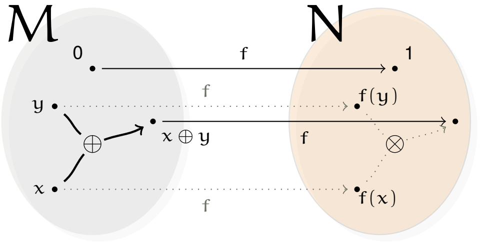
```
val stringMonoid: Monoid[String] =
  new Monoid[String] {
    def op (a1: String, a2: String): String =
      a1 + a2
    val zero: String = ""
  }

...
class MonoidSpec extends ...
  "stringMonoid is a monoid" in
    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 $\quad f(x \oplus y) = f(x) \otimes f(y)$ for any $x, y \in M$

Preserves identity $\quad f(0) = 1$

Distributive: `forAll { (x:A, y: A) => f (M.op (x,y)) should be (N.op (f (x), f (y))) }`

Preserves identity: `f (M.zero) should be (N.zero)`

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

- Understand the type class and the interface
- Foldable is a higher kind, parameterized with a type constructor
- Folds work with Monoids operator, no diff btw left and right
- Reduce = a fold from monoid's identity with op
- Mapreduce (spark) can distribute calculations in monoids
- Behavior interface, like Enumerable, Iteratable, etc. – (Monoid could be called "Addable")
- Mentimeter ($\times$2)

```
trait Foldable[F[_]] {

  def foldRight[A,B] (as: F[A]) (z: B) (f: (A,B) => B): B

  def foldLeft[A,B] (as: F[A]) (z: B) (f: (B,A) => B): B

  def foldMap[A,B] (as: F[A]) (f: A => B) (mb: Monoid[B]): B

  def concatenate[A] (as: F[A]) (m: Monoid[A]): A =
    foldLeft (as) (m.zero) (m.op)
}
```

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

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

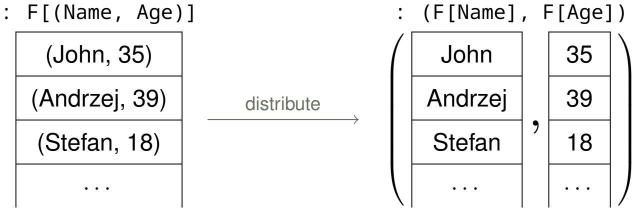
- Understand the type class and the interface
- Functor is a higher kind, parameterized with a type constructor
- The law: structure preservation
- Why 'self'? Why the implicit constraint?
- Even though `map` is a weak assumption we can derive useful functions (distribute)
- Distribute is a general unzipper for any functor, not just lists
- Distribute is explained in the next slide
- General zippers require `map2` (a Monad, or an Applicative Functor—not in curriculum)

```scala
trait Functor[F[_]] { self =>

  def map[A,B] (fa: F[A]) (f: A => B) :F[B]

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

  object Laws {
    def map[A] (implicit arbFA: Arbitrary[F[A]]) =
      forAll { fa: F[A] =>
        self.map[A, A] (fa) (identity[A]) should be (fa) }
```

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

Functor

Distribute




```scala
def distribute[A,B] (fab: F[(A,B)]): (F[A],F[B]) =
  (map (fab) (_._1), map (fab) (_._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.

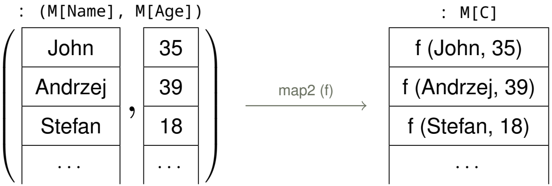
Exercise: Doodle an instance for `Option` (go back to the previous slide)

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] =
  flatMap (fa) (a => map (fb) (b => f (a,b)))
def map2[A,B,C] (fa: List[A], fb: List[B]) (f: (A,B) => C): List[C] =
  flatMap (fa) (a => map (fb) (b => f (a,b)))
def map2[A,B,C] (fa: Gen[A], fb: Gen[B]) (f: (A,B) => C): Gen[C] =
   flatMap (fa) (a => map (fb) (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` (values incorrect)
- What happens for `List` ?
- What happens for `Option` ?
- Parser is a bit similar to `Option` and `Gen`

Monad

Type Class

- 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

```scala
trait Monad[F[_]] extends Functor[F] { self =>

  def unit[A]  (a: => A): F[A]
  def flatMap[A,B] (ma: F[A]) (f: A => F[B]): F[B]

  // derived
  def map[A, B] (ma: F[A]) (f: A => B): F[B] =
    flatMap (ma) (a => unit (f (a)))
  def map2[A, B, C] (ma: F[A], mb: F[B]) (f: (A,B) => C): F[C] =
    flatMap (ma) (a => map (mb) (b => f (a, b)))
```

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

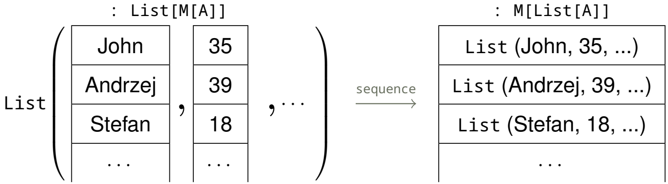
- Laws for flatMap and unit: associative, identity
- Discussion of type constraints, where is `arbAFA` used?
- This is formulated for Scalatest (imperative) not for Scalacheck!

```
def associative[A, B, C] (implicit arbFA: Arbitrary[F[A]],
                                   arbAFB: Arbitrary[A => F[B]],
                                   arbBFC: Arbitrary[B => F[C]]) =
  forAll { (x: F[A], f: A => F[B], g: B => F[C]) =>
    val left = self.flatMap (self.flatMap (x) (f)) (g)
    val right = self.flatMap (x) (a => self.flatMap (f (a)) (g))
    left should be (right)
  }

def identityRight[A]
  (implicit arbFA: Arbitrary[F[A]], arbAFA: Arbitrary[A => F[A]]) =
  forAll { (x: F[A], f: A => F[A]) =>
    self.flatMap[A, A] (x) (a => self.unit[A] (a)) should be (x) }

def identityLeft[A: Arbitrary] (implicit arbAFA: Arbitrary[A => F[A]]) =
  forAll { (y: A, f: A => F[A]) =>
    self.flatMap[A, A] (self.unit[A] (y)) (f) should be (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` (values incorrect)
- What happens for `Option` ?
- What happens for `List` ?
- Parser is a bit similar to `Option` and `Gen`