6 — Foldables and Functors

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2023

Lehrstuhl für Informatik VI, Uni Würzburg

General monoid folding

Last time we looked at monoids. We already noticed, that their parts fit the required parameters for folds.

We will take a closer look at folding with monoids and see a third fold beside `foldLeft` and `foldRight` only for monoids.

Exercise: General monoid folding

Last lecture we saw the method to fold a list with a monoid:

```scala
def combineAll[A](as: List[A])(using m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.combine)
```

If we don't have a monoid instance for the type, we can map to a monoidal type first. But can it be done in a single iteration?

Implement `foldMap` with a single `foldLeft` (no `map` and `combineAll`)

```scala
def foldMap[A,B](as: List[A])(f: A => B)(using m: Monoid[B]): B
```

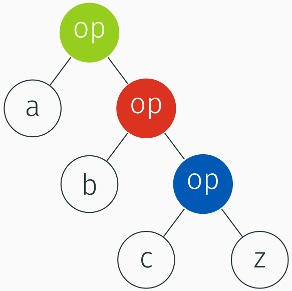
Exercise: General monoid folding — Solution

```scala
def foldMap[A,B](as: List[A])(f: A => B)(using m: Monoid[B]): B =
  as.foldLeft(m.zero)((b, a) => m.combine(b, f(a)))
```

Fold associativity

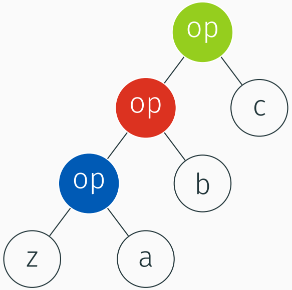
Recall: difference between left and right fold is associativity

Right fold:



```
op(a, op(b, op(c, z)))
```

Left fold:



```
op(op(op(z, a), b) ,c)
```

Folding with monoids

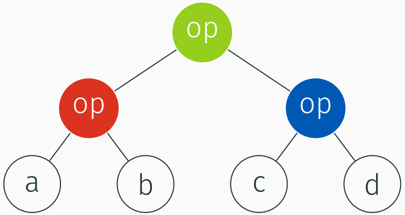
```
val words = List("Lorem", "Ipsum", "Dolor")
// words: List[String] = List(Lorem, Ipsum, Dolor)
val s = words.foldRight(stringMonoid.zero)(stringMonoid.combine)
// s: String = "LoremIpsumDolor"
val t = words.foldLeft(stringMonoid.zero)(stringMonoid.combine)
// t: String = "LoremIpsumDolor"
```

Because of associativity and identity laws of monoids, foldLeft and foldRight have same result:

```
//         left fold                                    right fold
(("" + "Lorem") + "Ipsum") + "Dolor" == "Lorem" + ("Ipsum" + ("Dolor" + ""))
```

Balanced fold

With monoids, which are associative, a balanced fold is possible too:



```
op(op(a, b), op(c, d))
```

- can be parallelized
- can be more efficient, if cost of **op** is proportonal to size of arguments (e.g. string concatenation)

Balanced foldMap

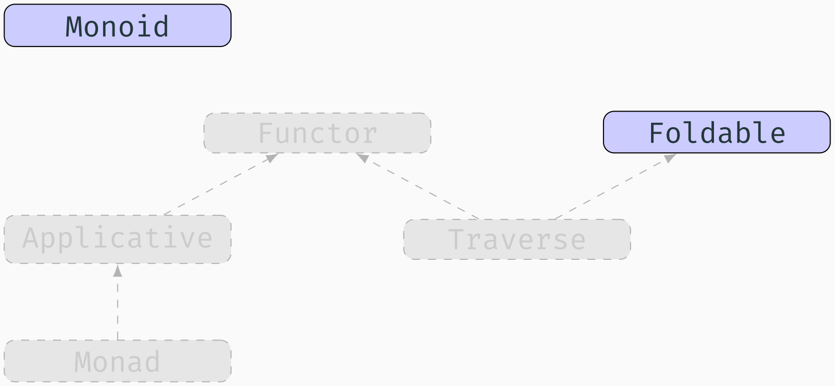
```
def foldMapBalanced[A,B](as: Vector[A])(f: A => B)(using m: Monoid[B]): B =
  if as.length == 0 then
    m.zero
  else if as.length == 1 then
    f(as(0))
  else
    val (left, right) = as.splitAt(as.length / 2)
    m.combine(foldMapBalanced(left)(f), foldMapBalanced(right)(f))
```

We use `Vector` here, because it has more effiecient splitting and length operations.

This implementation is not parallelized, but we would only have to change the last line to have the recursive calls in different threads to do so (e.g. using `Future`, out of scope here).

Abstracting folds

Typeclass hierarchy


$$\begin{tikzcd}
\text{Monoid} & & & \\
& \text{Functor} & & \text{Foldable} \\
\text{Applicative} \arrow[ur, dashed] & & \text{Traverse} \arrow[ul, dashed] \arrow[ur, dashed] & \\
\text{Monad} \arrow[u, dashed] & & &
\end{tikzcd}$$

Abstracting folds

We have seen several structures, that can be folded:

- Lists
- LazyLists
- Vectors
- Option (we can treat it like a list with at most one element)
- (in the exercise we'll have another one: Trees)

Given e.g. a structure of ints that we want to sum up:

```
ints.foldRight(0)(_ + _)
```

type of `ints` irrelevant for us, only existence of `foldRight`.

Foldable

We can find the common parts between all our foldable structures. As the signatures of the folds themselves don't even contain the type they work on any more, we can copy them unchanged:

```
trait Foldable[F[_]]:
  extension [A](as: F[A])
    def foldRight[B](z: B)(f: (A,B) => B): B

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

    def foldMap[B](f: A => B)(using mb: Monoid[B]): B =
      foldLeft(mb.zero)((b, a) => mb.combine(b, f(a)))

    def combineAll(m: Monoid[A]): A =
      foldLeft(m.zero)(m.combine)
```

Here, `F[_]` denotes a *type constructor* (more on next slide).

Higher kinded types

The parameter definition of `F[_]` means, `F` is not a concrete type, but a type constructor taking one argument.

We already know values have types. But types also have types, which are called kinds. Kinds basically tell us, what kind and amount of types have to be provided to get to a concrete type. Some examples of kinds:

	Example	Kind
Concrete type	`String`, `List[Int]`	$*$
TC, one param	`List`, `Option`	$* \to *$
TC, two params	`Either`	$* \to * \to *$
TC, takes TC as param	`Foldable`	$(* \to *) \to *$

Kinds with arrows are similar to functions, but on the type level.

Higher kinded types

`Foldable` can take a type constructor of kind $* \to *$ as parameter, which in Scala is a type with one type parameter: (Note: no brackets after `List`)

```
given Foldable[List] with
  // ...
```

As seen in fold signatures, `F` can be used with parameters:

```
trait Foldable[F[_]]:
  extension [A](as: F[A])
    def foldRight[B](z: B)(f: (A,B) => B): B
```

Implementation for `List` would have a `List[A]` as the type on which extensions are added.

`Foldable` is a type constructor that takes a type constructor as parameter, a.k.a. higher-order type constructor or higher-kinded type (cf. higher order function).

Higher kinded types

With this, we can add a modified higher-kind version of our monoid:

```scala
trait MonoidK[F[_]]:
  def zero[A]: F[A]
  def combine[A](f1: F[A], f2: F[A]): F[A]
```

which allows us to have a single instance for types like `List` or `Option`, where the element type is not important for combine and zero, e.g. for Option:

```scala
given MonoidK[Option] with
  def zero[A]: Option[A] = None
  def combine[A](o1: Option[A], o2: Option[A]): Option[A] = o1 orElse o2
```

The variant from last time generated one instance per element type, so this is better.

Exercise: Using Foldable

We already defined `foldLeft` and `foldRight` for List, etc., so creating the `Foldable` instances is not that interesting. So let's try to use the typeclass instead!

Write a method that can turn any foldable data structure into a list. Think about how the signature of this method should look.

Hint: the method will need two type parameters of different kinds.

Hint: we've already seen the implementation in the lecture on lists.

Using Foldables — Solution

```scala
def toList[F[_], A](data: F[A])(using Foldable[F]): List[A] =
  data.foldRight[List[A]](Nil)((a, res) => a :: res)
```

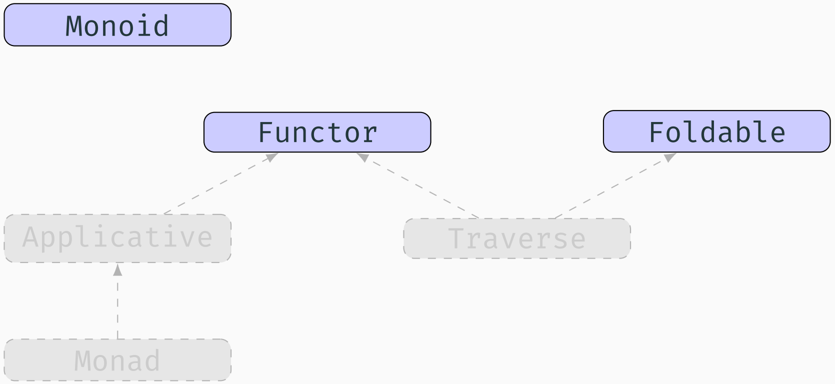
Foldable is of kind $(* \to *) \to *$, so it needs a parameter of kind $* \to *$. Therefore our function does too so it can pass it on: `F[_]`.
For our folded data structure, we additionally need the element type (kind $*$) to pass to `F[_]`, so we can get a concrete type: `A`.

With those we can get `F[A]`, a type we can use the methods from a `Foldable[F]` instance on.

Like seen in the list lecture, `foldRight` is perfect for creating a list.

Functors

Typeclass hierarchy



A New Take on Old Comrades

Option:

```
def getSalary(p: Person): Option[Int] =
  for                                                    // may fail if:
    id         <- getPersonId(p)                         // - person not in DB
    baseSalary <- getPosition(id).map(_.baseSalary) // - id is not in DB
    bonus      <- getBonus(id)                           // - id is not in DB
  yield baseSalary + bonus
```

Which is basically equivalent to

```
def getSalary(p: Person): Option[Int] =
  getPersonId(p).flatMap(id =>
      getPosition(id).map(_.baseSalary).flatMap(baseSalary =>
          getBonus(id).map(bonus => baseSalary + bonus)))
```

Option allows you to handle errors or the absence of values and fail fast.

A New Take on Old Comrades

Functions:

```scala
val double: (Int => Int) = i => i * 2

val asHex: (Int => String) = i => "0x" + Integer.toString(i, 16)

val bytes: (String => List[Byte]) = _.getBytes.toList

val combined: (Int => List[Byte]) = double.andThen(asHex).andThen(bytes)
```

Functions can be chained, always keeping the input type of the first function.

A New Take on Old Comrades

Function and **Option** do completely different things and share practically no behavioural similarity. But not all is different. Lets only look at functions taking an Int:

```
type IntFunction[A] = Int => A
```

```
def     map[A, B](a:      Option[A])(f: A => B):      Option[B]
def andThen[A, B](a: IntFunction[A])(f: A => B): IntFunction[B]
```

`Option`'s `map` and `IntFunction`'s `andThen` have a pretty similar signature.

The same is true for `Either` and `List`, which also have a corresponding `map` function.

The Functor

There seems to a pattern there. Let’s try to capture that.

```scala
trait Functor[F[_]]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B]
```

So a `Functor` is a unary type constructor which allows us to map over it.

We also see now, why we created the `IntFunction` alias: `F[_]` here expects a single parameter i.e. F’s kind is $* \rightarrow *$, while a unary function has kind $* \rightarrow * \rightarrow *$.

The Functor

Implement the `Functor` for `Option`.

```
trait Functor[F[_]]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B]
```

Solution:

```
given Functor[Option] with
  extension [A](fa: Option[A])
    def map[B](f: A => B): Option[B] =
      fa match
        case None    => None
        case Some(a) => Some(f(a))
```

or simply use the existing `map`

```
def map[B](f: A => B): Option[B] = fa.map(f)
```

The Functor

We can now define functions purely in terms of functor. Adding them to the trait makes them available on all types with functor instances.

```
//in trait Functor
extension [A](fa: F[A])
  def map[B](f: A => B): F[B]

  /* map, but also keep old value */
  def fproduct[B](f: A => B): F[(A, B)] =
    fa.map(a => (a, f(a)))

extension [A,B](fab: F[(A, B)])
  /* split a functor with a tuple */
  def unzip: (F[A], F[B]) =
    (map(fab)(_._1), map(fab)(_._2))
```

Both of these new functions "just work" as long as there is a `map` method.

The Functor — Laws

To ease reasoning about them we also give those algebraic abstractions some laws. We already saw the identity law:

```
y.map(id) == y
```

Thanks to parametricity, a lot follows from there.

Discussion:

- Can map reorder a list?
- Can map drop elements from a list?

The Functor — Laws

To ease reasoning about them we also give those algebraic abstractions some laws. We already saw the identity law:

```
y.map(id) == y
```

Thanks to parametricity, a lot follows from there.

Discussion:

- Can map reorder a list?
- Can map drop elements from a list?

No! This law forces every functor to keep its structure and just replace the elements.

The Functor — Laws

The composition law states, that if we map with one function and then the other, the result should be the same as mapping with the composition of both functions:

```
x.map(f).map(g) == x.map(f andThen g)
```

Composing functors

Similar to monoids, functors can be composed. We can nest any number of them and use `map` to modify the value in the innermost functor. E.g. if we have an `Option[List[Int]]`, we may want to run a calculation on all the ints in the list, without first unwrapping it.

To define this composition, we first need type lambdas, as `F[G[_]]` would be interpreted by Scala as passing `G[_]` as parameter to F, so passing a $* \to *$ to a $* \to *$, causing a type error.

Composing functors — Type Lambdas

We want F[G[_]] to be interpreted as a combined type of kind $* \to *$. With Function we used a type alias, which is possible but clumsy. We can als define a type lambda directly where we want to use the type:

```
Functor[   [a] =>> F[G[a]]   ]
//          ↑          ↑
//   type parameter    use of parameter
```

This is pretty similar to lambdas on the value level, but using brackets instead of parentheses. The variant with _is similar to the underscore shorthand for value lambdas.

Composing functors

With this, we can now create a method that combines two functors:

```scala
given [F[_],G[_]](using FF: Functor[F], FG: Functor[G])
: Functor[[a] =>> F[G[a]]] with
  extension [A](fga: F[G[A]])
    def map[B](f: A => B): F[G[B]] = FF.map(fga)(ga => FG.map(ga)(f))
```

With this, we can map the `A` in an `F[G[A]]` directly, if both `F` and `G` have a functor instance.

Composing functors

Because `F` also has a functor, when we want to treat `F[G[_]]` as a composed functor we have to do it explicitly:

```scala
def myfunc[F[_]](someInt: F[Int])(using Functor[F]) = someInt.map(_ + 1)
val nested: Option[List[Int]] = Some(List(1,2,3))
```

```scala
/* error  Required: F[Int]
          where:    F is a type variable with constraint <: [_] =>> Any */
myfunc(nested)
```

```scala
// works
myfunc[[a] =>> Option[List[a]]](nested)
```

Conclusion

We have seen today:

- that folds with an associative operator can be split more efficiently
- how to describe the “type of a type” with kinds, so we can handle type constructors and type lambdas without a concrete type
- how to use this to create a higher kinded monoid and abstract over structures that can be folded (`Foldable`) or mapped (`Functor`)
- the Functor laws, and that nested Functors can also be treated as a single Functor.

Exercise, if there is time left: Creating the `Show` type class from scratch

The Show type class

On the JVM, every object has a `toString` method, which may or may not be defined in a useful way. If it is not overridden, it just shows a JVM internal object id.

The `Show` type class fulfills the same purpose, but is not automatically available on any type, but only on the ones which actually want to provide a text representation.

The Show type class

Define a type class named `Show` with a single type parameter `A`.

It should have a single extension method for `A` named `show`, taking no parameters and returning a `String`.

The Show type class — Solution

```scala
trait Show[A]:
  extension (a: A)
    def show: String
```

Side note: can also be defined with empty parentheses, i.e. `def show()`, but then calling it also requires them. Usually parentheses for parameterless methods are used to signify side effects in Scala (by convention)

Show instances

Create a `Show` instance for the case class `Person`:

```scala
case class Person(lastName: String, firstName: String, age: Int)
```

It should create a string of the form `"<firstName> <lastName> is <age> years old"`, e.g. given the value

```scala
val janedoe = Person("Doe", "Jane", 42)
```

this should be true:

```scala
janedoe.show == "Jane Doe is 42 years old"
```

Think about where this should be placed.

Show instances — Solution

Some possible variants of how to implement this:

```scala
given Show[Person] with
  extension (a: Person)
    def show: String =
      s"${a.firstName} ${a.lastName} is ${a.age} years old"
```

```scala
given Show[Person] with
  extension (a: Person)
    def show: String =
      a.firstName + " " + a.lastName + " is " + a.age.toString + " years old"
```

Usually placed on the companion object of the type or the typeclass, if possible.