Object Meets Function

The Power of HoF

Steven Lolong

steven.lolong(at)uni-tuebingen.de

Programming Language Research Group
Tuebingen University

Nov. 23, 2022

The Power of HoF

1. Map
2. Filter
3. Fold
4. Zip

Eberhard Karls
Universität
Tübingen

Mapping [Functor]

Borrow the concept from category theory

A Functor is a mapping between categories.

A Category is a collection of "object" that are linked by "arrow".

One of a good resources: **Bartosz Milewski**. **Category Theory for Programmers**.

A functor is simply a container. Given a container, and a function which works on the elements, we can apply that function to each element (wiki.haskell.org).

Mapping [Functor]

fmap : **(object-A) (fArrow) => object-B**

- object-A the source object.
- fArrow the function that will map the data of source object (object-A) to become a new data of target object (object-B).
- object-B the target object.

Mapping [Functor]

Let's do it!

```
1 def fMap(dList: List[Int])(fOptr:(Int) => Int) : List[Int] =
      dList match
2   case Nil => Nil
3   case (x :: xs) => fOptr(x) :: fMap(xs)(fOptr)
4
5 val exList = List(1,2,3,4,5)
6
7 fMap(exList)((a) => a + 2)
```

Listing: FMap - Example1

Mapping [Functor]

Let's do it!

How about if the object's type is varied.

Mapping [Functor]

Let's do it!

```
1 def fMapTP1 [A](dList: List[A]) (fOptr: A => A) : List[A] =
      dList match
2   case Nil => Nil
3   case (x :: xs) => fOptr(x) :: fMapTP1(xs)(fOptr)
```

Listing: FMap - Example2

Mapping [Functor]

Let's do it!

How about if the source object's type is different with the target object's type.

Mapping [Functor]

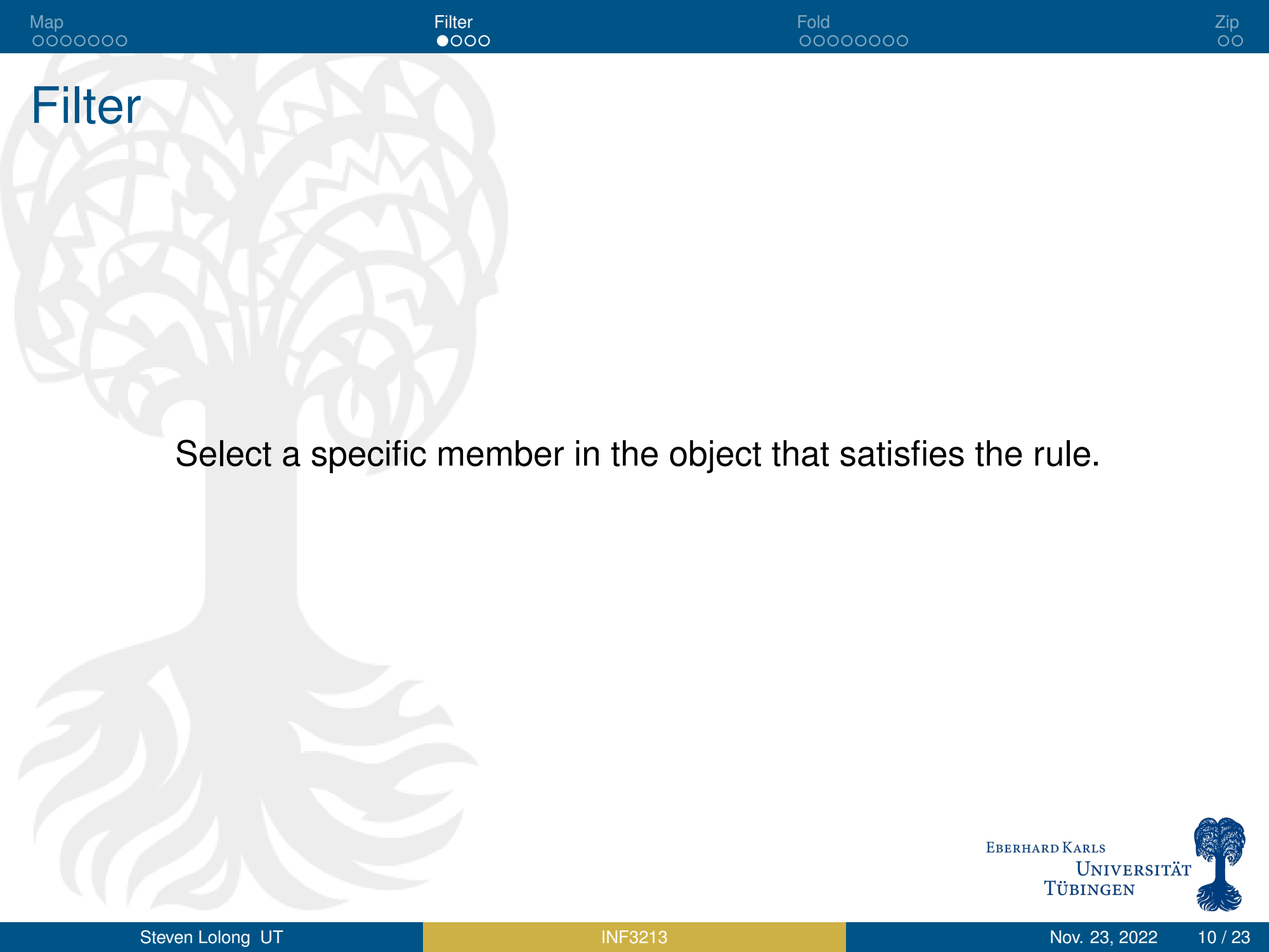
Let's do it!

```
1 def fMapTP [A,B](dList: List[A]) (fOptr: A => B) : List[B] =
      dList match
2   case Nil => Nil
3   case (x :: xs) => fOptr(x) :: fMapTP(xs)(fOptr)
```

Listing: FMap - Example3

Filter

Select a specific member in the object that satisfies the rule.



Filter

filter:**(object-A)(fRule: Boolean) => object-B**

Filter

```
1  def filter (dList: List[Int])(fFilter: Int => Boolean) :
       List[Int] = dList match
2    case Nil => Nil
3    case (x :: xs) => fFilter(x) match
4      case true => x :: filter(xs)(fFilter)
5      case false => filter(xs)(fFilter)
6
7  val exList = List(1,2,3,4,5,6)
8
9  val dEven = filter(exList)((a) => (a % 2) match
10   case 0 => true
11   case _ => false)
```

Listing: Filter - Example1

Filter

```
1 def filterTP [A](dList: List[A]) (fFilter: (A) => Boolean) :
      List[A] = dList match
2   case Nil => Nil
3   case (x :: xs) => fFilter(x) match
4     case true => x :: filterTP(xs)(fFilter)
5     case false => filterTP(xs)(fFilter)
```

Listing: Filter - Example2

Eberhard Karls Universität Tübingen

Fold

Fold (Reduce) is a family of higher order functions that process a data structure in some order and build a return value.

Fold

Right

foldr:**(object-A)(reducer: A => B) => object-B**

Fold

Right

```
1 def rFold(dList: List[Int])(dIdent: Int)(f: (a:Int, b:Int) =
    > Int) : Int = dList match
2   case Nil => dIdent
3   case (x :: xs) =>  f(x, rFold(xs)(dIdent)(f))
4
5 val exList = List(1,2,3,4,5)
6
7 rFold(exList)(0)(_ + _)
```

Listing: Foldr - Example1

Fold

Right

```
1 def rFoldTV1 [A](dList: List[A])(dIdent: A) (f: (A, A) => A)
      : A = dList match
2   case Nil => dIdent
3   case (x :: xs) => f(x, rFoldTV1(xs)(dIdent)(f))
```

Listing: Foldr - Example2

Fold

Right

```
1 def rFoldTV2 [A,B](dList: List[A])(dIdent: B) (f: (A, B) =>
      B) : B = dList match
2   case Nil => dIdent
3   case (x :: xs) => f(x, rFoldTV2(xs)(dIdent)(f))
```

Listing: Foldr - Example3

Fold

Left

```
1  def lFold(dList : List[Int])(accul: Int)(f: (Int, Int) =>
       Int) : Int = dList match
2    case Nil => accul
3    case (x :: xs) => lFold (xs) (f(x, accul)) (f)
4
5  val exList = List(1,2,3,4)
6
7  lFold (exList)(0)(_ + _)
8
9  // Build factorial function base on lFold
10 val factorialFunc = (a: Int) => a match
11   case x if x <= 1 => 1
12   case _ => lFold(List.range(1,(a + 1)))(1)(_ * _)
```

Listing: Foldl - Example1

Fold

Left

```
1   case x if x <= 1 => 1
2   case _ => lFold(List.range(1,(a + 1)))(1)(_ * _)
```

Listing: Foldl - Example2

Fold

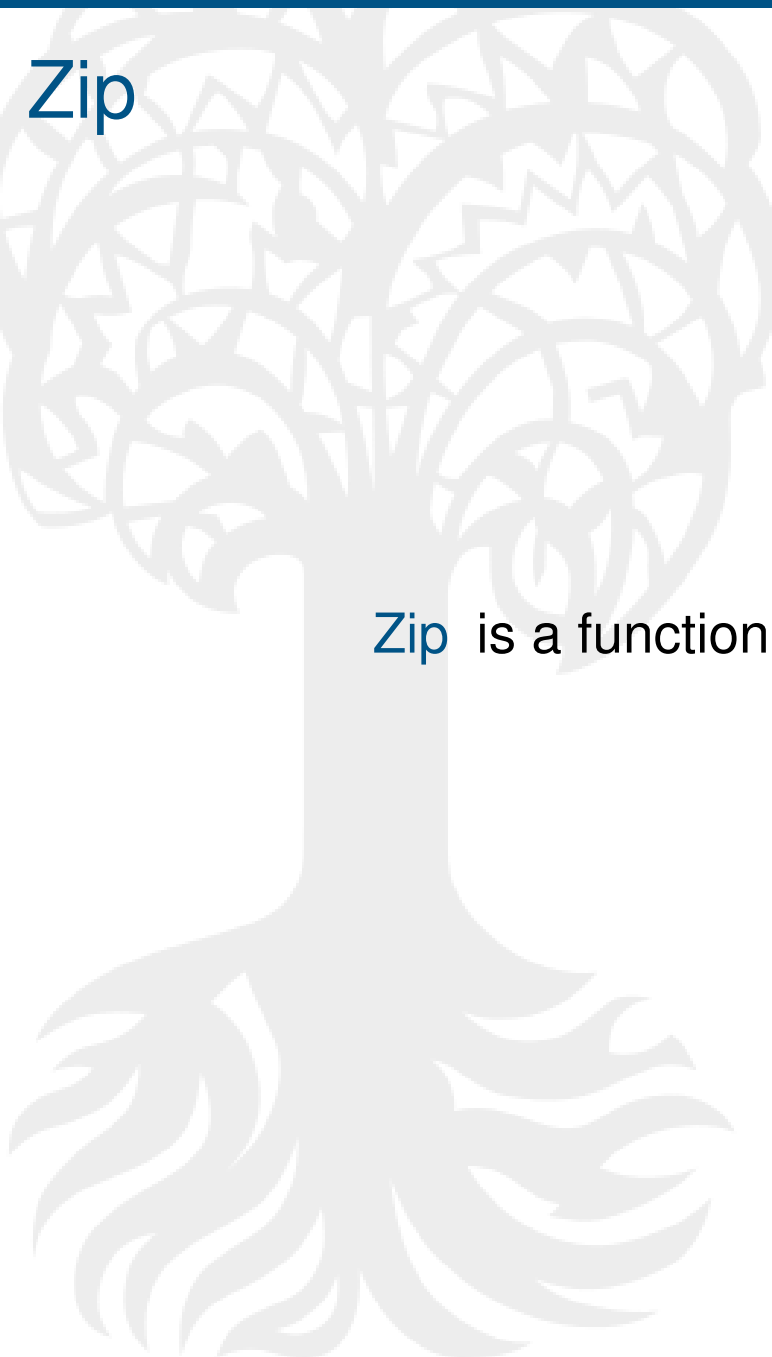
Left

```
1 def lFoldTV1[A](dList: List[A])(accul: A)(f: (A,A) => A) : A
      = dList match
2   case Nil => accul
3   case (x :: xs) => lFoldTV1(xs)(f(accul,x))(f)
```

Listing: Foldl - Example3

Zip

Zip is a function that combines two objects into one.



Fold

Left

```
1  def zipIt[A,B](a: List[A])(b: List[B]) : List[(A,B)] = (a,b)
       match
2    case (Nil, _) => Nil
3    case (_, Nil) => Nil
4    case ((x :: xs), (y :: ys)) => (x,y) :: zipIt(xs)(ys)
5
6  val listA = List.range(1,5)
7  val listB = List.range(4,7)
8
9  zipIt[Int,Int] (listA)(listB)
10
11
12 zipIt (listA)(listB)
```

Listing: Zip - Example1