Functional Programming

Functors, Applicatives, and Parsers

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

Introduction

- Functors and applicatives are concepts from category theory
- A very general and abstract theory about structures and maps between them
- So general that mathematicians call it "general abstract nonsense"
- Yields very useful abstractions for functional programming
- After a brief review we specialize for Haskell

Plan

Categories

Definition (part 1)

A small category $\mathcal{C}$ is given by

- a set of objects,
- for each pair of objects A, B, a set $Hom(A, B)$ of arrows (morphisms) between A and B,
- for each pair of arrows $f \in Hom(A, B)$ and $g \in Hom(B, C)$ (for objects A, B, C), there is an arrow $(f ; g) \in Hom(A, C)$, the composition of f and g (alternatively), write $g \circ f$).

Moreover, the following laws are expected to hold

(Small) Categories

Definition (part 2: laws)

- For each object A there is a designated identity arrow $i_A \in Hom(A, A)$ which behaves as an identity with respect to composition:
  - for each $f \in Hom(A, B)$, $i_A; f = f$,
  - for each $g \in Hom(B, A)$, $g; i_B = g$.
- Composition of arrows is associative, that is:

$$f; (g; h) = (f; g); h$$

for all $f \in Hom(A, B)$, $g \in Hom(B, C)$, and $h \in Hom(C, D)$ and objects A, B, C, D.

Examples of categories (not small)

Set

Objects are sets and morphisms are total functions.

Examples of categories (not small)

Set

Objects are sets and morphisms are total functions.

Par

Objects are sets, morphisms are partial functions.

Examples of categories (not small)

Set

Objects are sets and morphisms are total functions.

Par

Objects are sets, morphisms are partial functions.

Group, **Ring**, **Vect**

Objects are groups (rings, vector spaces), morphisms are group (ring, vector space) homomorphisms

Smaller categories

FinSet (only essentially small)

Objects are finite sets and morphisms are total functions.

Smaller categories

FinSet (only essentially small)

Objects are finite sets and morphisms are total functions.

Partially ordered sets

Every poset $(A, \leq)$ gives rise to a category with objects $a \in A$ and a single morphism m_{ab} for each $a, b \in A$ such that $a \leq b$.

Smaller categories

FinSet (only essentially small)

Objects are finite sets and morphisms are total functions.

Partially ordered sets

Every poset $(A, \leq)$ gives rise to a category with objects $a \in A$ and a single morphism m_{ab} for each $a, b \in A$ such that $a \leq b$.

Graphs

Every directed graph (N, E) gives rise to category with objects $n \in N$ and morphisms paths in N.

Smaller categories

FinSet (only essentially small)

Objects are finite sets and morphisms are total functions.

Partially ordered sets

Every poset $(A, \leq)$ gives rise to a category with objects $a \in A$ and a single morphism m_{ab} for each $a, b \in A$ such that $a \leq b$.

Graphs

Every directed graph (N, E) gives rise to category with objects $n \in N$ and morphisms paths in N.

Hask (small?)

Objects are Haskell types, morphisms are Haskell functions.

Plan

Functors (in general)

Definition

Suppose $\mathcal{C}$ and $\mathcal{D}$ are categories. A functor $F : \mathcal{C} \to \mathcal{D}$ consists of

- a mapping from objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and
- a mapping from arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$

such that

- $f \in HOM_{\mathcal{C}}(A, B)$ is mapped to $F(f) \in HOM_{\mathcal{D}}(FA, FB)$,
- $i_A \in HOM_{\mathcal{C}}(A, A)$ is mapped to $i_{FA} \in HOM_{\mathcal{D}}(FA, FA)$,
- $f \in HOM_{\mathcal{C}}(A, B)$ and $g \in HOM_{\mathcal{C}}(B, C)$ implies that $F(f; g) = F(f); F(g) : HOM_{\mathcal{D}}(FA, FC)$

for all objects A, B, C of $\mathcal{C}$.

Functors (in general)

Definition

Suppose $\mathcal{C}$ and $\mathcal{D}$ are categories. A functor $F : \mathcal{C} \to \mathcal{D}$ consists of

- a mapping from objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and
- a mapping from arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$

such that

- $f \in HOM_{\mathcal{C}}(A, B)$ is mapped to $F(f) \in HOM_{\mathcal{D}}(FA, FB)$,
- $i_A \in HOM_{\mathcal{C}}(A, A)$ is mapped to $i_{FA} \in HOM_{\mathcal{D}}(FA, FA)$,
- $f \in HOM_{\mathcal{C}}(A, B)$ and $g \in HOM_{\mathcal{C}}(B, C)$ implies that $F(f;g) = F(f); F(g) : HOM_{\mathcal{D}}(FA, FC)$

for all objects A, B, C of $\mathcal{C}$.

Remark

An Endofunctor on a category $\mathcal{C}$ is a functor from $\mathcal{C} \to \mathcal{C}$.

Functors (Endofunctors on Hask)

Definition

A functor is a mapping `f` between types such that for every pair of type `a` and `b` there is a function `fmap :: (a -> b) -> (f a -> f b)` such that the functorial laws hold:

1. the identity function on `a` is mapped to the identity function on `f a`:
   `fmap id fx == id fx`, for all `fx` in `f a`
2. `fmap` is compatible with function composition
   `fmap (f . g) == fmap f . fmap g`, for all `f :: b -> c` and `g :: a -> b`

Functors (Endofunctors on Hask)

Definition

A functor is a mapping f between types such that for every pair of type a and b there is a function fmap :: (a –> b) –> (f a –> f b) such that the functorial laws hold:

1. the identity function on a is mapped to the identity function on f a:
   fmap **id** fx == **id** fx, for all fx in f a
2. fmap is compatible with function composition
   fmap (f . g) == fmap f . fmap g, for all f :: b –> c and g :: a –> b

Functions on types

- **Int**, **Bool**, **Double** etc are types.
- parameterized types like [a], BTree a, **IO** a can be considered as a type constructor (i.e., [], BTree, **IO**) applied to a type
- We can express that formally by writing kindings: **Int** :: ∗, **Bool** :: ∗, **Double** :: ∗, but [] :: ∗ –> ∗, BTree :: ∗ –> ∗, **IO** :: ∗ –> ∗

Functors in Haskell

The functor class

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

- Recall `f` is a type variable that can stand for **type constructors** (ie, functions on types) like `IO`, `[]`, and others. So `f :: * -> *`!

Functors in Haskell

The functor class

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

- Recall `f` is a type variable that can stand for **type constructors** (ie, functions on types) like `IO`, `[]`, and others. So f :: * -> *!

Good news

We already know a couple of functors!

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor
- fmap :: (a -> b) -> (f a -> f b)

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor
- fmap :: (a -> b) -> (f a -> f b)
- fmap :: (a -> b) -> ([a] -> [b])

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor
- fmap :: (a −> b) −> (f a −> f b)
- fmap :: (a −> b) −> ([a] −> [b])
- Looks familiar?

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor
- fmap :: (a −> b) −> (f a −> f b)
- fmap :: (a −> b) −> ([a] −> [b])
- Looks familiar?
- It's the type of **map**

List is a functor

- To make list an instance of functor, we need to instantiate the type f in the type of fmap by [], the list type constructor
- fmap :: (a −> b) −> (f a −> f b)
- fmap :: (a −> b) −> ([a] −> [b])
- Looks familiar?
- It's the type of **map**
- It remains to check the functorial laws on **map**

Functorial laws for list

fmap id fx == id fx

fx is a list, so we must proceed by induction

- **map id** [] == [] == **id** []
- **map id** (x:xs) == **id** x : **map id** xs == x : xs == **id** (x : xs)

Functorial laws for list

fmap **id** fx == **id** fx

fx is a list, so we must proceed by induction

- **map id** [] == [] == **id** []
- **map id** (x:xs) == **id** x : **map id** xs == x : xs == **id** (x : xs)

fmap (f . g) == fmap f . fmap g

Must hold when applied to any list fx

- **map** (f . g) [] == [] == **map** f (**map** g [])
- **map** (f . g) (x : xs) == (f . g) x : **map** (f . g) xs
  == f (g x) : (**map** f . **map** g) xs by function composition and induction
  == f (g x) : **map** f (**map** g xs) by function composition
  == **map** f (g x : **map** g xs) by **map** f
  == **map** f (**map** g (x : xs)) by **map** g
  == (**map** f . **map** g) (x : xs)

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a −> b) −> (f a −> f b)

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a −> b) −> (f a −> f b)
- **mapMaybe** :: (a −> b) −> (**Maybe** a −> **Maybe** b)

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a −> b) −> (f a −> f b)
- **mapMaybe** :: (a −> b) −> (**Maybe** a −> **Maybe** b)
- There is actually no real choice for its definition

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a -> b) -> (f a -> f b)
- **mapMaybe** :: (a -> b) -> (**Maybe** a -> **Maybe** b)
- There is actually no real choice for its definition
- **mapMaybe** g **Nothing** = **Nothing**

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a -> b) -> (f a -> f b)
- **mapMaybe** :: (a -> b) -> (**Maybe** a -> **Maybe** b)
- There is actually no real choice for its definition
- **mapMaybe** g **Nothing** = **Nothing**
- **mapMaybe** g (**Just** a) = **Just** (g a)

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a −> b) −> (f a −> f b)
- **mapMaybe** :: (a −> b) −> (**Maybe** a −> **Maybe** b)
- There is actually no real choice for its definition
- **mapMaybe** g **Nothing** = **Nothing**
- **mapMaybe** g (**Just** a) = **Just** (g a)
- Second equation could return **Nothing**, but that would violate the functorial laws

Maybe is a functor

- Reminder: **data Maybe** a = **Nothing** | **Just** a
- To make **Maybe** an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor **Maybe**
- fmap :: (a −> b) −> (f a −> f b)
- **mapMaybe** :: (a −> b) −> (**Maybe** a −> **Maybe** b)
- There is actually no real choice for its definition
- **mapMaybe** g **Nothing** = **Nothing**
- **mapMaybe** g (**Just** a) = **Just** (g a)
- Second equation could return **Nothing**, but that would violate the functorial laws
- It remains to check the functorial laws on **mapMaybe**

Functorial laws for Maybe

fmap **id** fx == **id** fx

fx is a Maybe, so we must proceed by induction (cases)

- **mapMaybe id Nothing** == **Nothing** == **id Nothing**
- **mapMaybe id** (**Just** x) == **Just** x == **id** (**Just** x)

Functorial laws for Maybe

fmap id fx == id fx

fx is a Maybe, so we must proceed by induction (cases)

- **mapMaybe id Nothing == Nothing == id Nothing**
- **mapMaybe id (Just** x) == **Just** x == **id (Just** x)

fmap (f . g) == fmap f . fmap g

Must hold when applied to any Maybe fx

- **mapMaybe** (f . g) **Nothing == Nothing == map** f (**map** g **Nothing**)
- **mapMaybe** (f . g) (**Just** x)
  == **Just** ((f . g) x)
  == **Just** (f (g x)) by function composition
  == **mapMaybe** f (**Just** (g x)) by **map** f
  == **mapMaybe** f (**mapMaybe** g (**Just** x)) by **map** g
  == (**mapMaybe** f . **mapMaybe** g) (**Just** x)

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree
- fmap :: (a −> b) −> (f a −> f b)

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree
- fmap :: (a -> b) -> (f a -> f b)
- mapBTree :: (a -> b) -> (BTree a -> BTree b)

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree
- fmap :: (a −> b) −> (f a −> f b)
- mapBTree :: (a −> b) −> (BTree a −> BTree b)
- There is actually no real choice for its definition

```
mapBTree g Leaf = Leaf
mapBTree g (Node l a r) = Node (mapBTree g l) (g a) (mapBTree g r)
```

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree
- fmap :: (a -> b) -> (f a -> f b)
- mapBTree :: (a -> b) -> (BTree a -> BTree b)
- There is actually no real choice for its definition

```
mapBTree g Leaf = Leaf
mapBTree g (Node l a r) = Node (mapBTree g l) (g a) (mapBTree g r)
```

- In the second equation we need to transform the data at the node by g and the subtrees of type BTree a recursively to BTree b using the mapBTree function

BTree is a functor

- Reminder: **data** BTree a = Leaf | Node (BTree a) a (BTree a)
- To make BTree an instance of functor, we need to instantiate the type f in the type of fmap by the type constructor BTree
- fmap :: (a -> b) -> (f a -> f b)
- mapBTree :: (a -> b) -> (BTree a -> BTree b)
- There is actually no real choice for its definition

```
mapBTree g Leaf = Leaf
mapBTree g (Node l a r) = Node (mapBTree g l) (g a) (mapBTree g r)
```

- In the second equation we need to transform the data at the node by g and the subtrees of type BTree a recursively to BTree b using the mapBTree function
- It remains to check the functorial laws on mapBTree, but we'll leave this inductive proof to you.

Remark

- Many of the predefined type constructors have `Functor` instances
- Some of them may be unexpected
- For instance **instance Functor** ((,) a) makes the pair type into a functor by defining `fmap` on the second component
- Mapping on the first component would also define a (different) functor!
- (There are also functors with more than one argument. They have to fulfill the functorial laws in all arguments. The pair type constructor (,) is an example of a binary functor.)

Plan

Applicatives

- An applicative (functor) is a special kind of functor
- It has further operations and laws
- We motivate it with a couple of examples

Applicative

Example 1: sequencing IO commands

```haskell
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (io:ios) = do x <- io
                       xs <- sequence ios
                       return (x:xs)
```

Applicative

Example 1: sequencing IO commands

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (io:ios) = do x <- io
                       xs <- sequence ios
                       return (x:xs)
```

Alternative way

```
sequence [] = return []
sequence (io:ios) = return (:) `ap` io `ap` sequence ios

return :: Monad m => a -> m a
ap :: Monad m => m (a -> b) -> m a -> m b
```

Applicative

Example 2: transposition

```
transpose :: [[a]] -> [[a]]
transpose [] = repeat []
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

Applicative

Example 2: transposition

```
transpose :: [[a]] -> [[a]]
transpose [] = repeat []
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

Rewrite

```
transpose [] = repeat []
transpose (xs:xss) = repeat (:) `zapp` xs `zapp` transpose xss

zapp :: [a -> b] -> [a] -> [b]
zapp fs xs = zipWith ($) fs xs
```

Applicative Interpreter

A datatype for expressions

```
data Exp v
  = Var v -- variables
  | Val Int -- constants
  | Add (Exp v) (Exp v) -- addition
```

Applicative Interpreter

A datatype for expressions

```haskell
data Exp v
  = Var v -- variables
  | Val Int -- constants
  | Add (Exp v) (Exp v) -- addition
```

Standard interpretation

```haskell
eval :: Exp v -> Env v -> Int
eval (Var v) env = fetch v env
eval (Val i) env = i
eval (Add e1 e2) env = eval e1 env + eval e2 env

type Env v = v -> Int
fetch :: v -> Env v -> Int
fetch v env = env v
```

Applicative Interpreter

Alternative implementation

```
eval' :: Exp v -> Env v -> Int
eval' (Var v) = fetch v
eval' (Val i) = const i
eval' (Add e1 e2) = const (+) `ess` (eval' e1) `ess` (eval' e2)

ess a b c = (a c) (b c)
```

Applicative

Extract the common structure

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative

Laws

- Identity

```
pure id <*> v == v
```

- Composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

- Homomorphism

```
pure f <*> pure x = pure (f x)
```

- Interchange

```
u <*> pure y = pure ($ y) <*> u
```

Instances of Applicative

- List, Maybe, and IO are also applicatives

Instances of Applicative

- List, Maybe, and IO are also applicatives

Lists

```haskell
instance Applicative [] where
  -- pure :: a -> [a]
  pure a = [a]
  -- (<*>) :: [a -> b] -> [a] -> [b]
  fs <*> xs = concatMap (\f -> map f xs) fs
```

Instances of Applicative

- List, Maybe, and IO are also applicatives

Lists

```
instance Applicative [] where
  -- pure :: a -> [a]
  pure a = [a]
  -- (<*>) :: [a -> b] -> [a] -> [b]
  fs <*> xs = concatMap (\f -> map f xs) fs
```

Maybe

```
instance Applicative Maybe where
  -- pure :: a -> Maybe a
  pure a = Just a
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  Just f <*> Just a = Just (f a)
  _ <*> _ = Nothing
```

Plan

An interesting example for Applicatives

Simple arithmetic expressions

```
data Term = Con Integer
          | Bin Term Op Term
            deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
            deriving (Eq, Show)
```

An interesting example for Applicatives

Simple arithmetic expressions

```haskell
data Term = Con Integer
          | Bin Term Op Term
            deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
            deriving (Eq, Show)
```

Task: Parsing expressions

- Read a string like `"3+42/6"`
- Recognize it as a valid term
- Return `Bin (Con 3) Add (Bin (Con 42) Div (Con 6))`

Parsing

The type of a simple parser

```
type Parser token result = [token] -> [(result, [token])]
```

Combinator parsing

Primitive parsers

```
pempty :: Parser t r
succeed :: r -> Parser t r
satisfy :: (t -> Bool) -> Parser t t
msatisfy :: (t -> Maybe a) -> Parser t a
lit :: Eq t => t -> Parser t t
```

Combinator parsing II

Combination of parsers

```
palt :: Parser t r -> Parser t r -> Parser t r
pseq :: Parser t (s -> r) -> Parser t s -> Parser t r
pmap :: (s -> r) -> Parser t s -> Parser t r
```

A taste of compiler construction

A lexer

A lexer partitions the incoming list of characters into a list of tokens. A token is either a single symbol, an identifier, or a number. Whitespace characters are removed.

Underlying concepts

Parsers have a rich structure

- parsing illustrates functors, applicatives, as well as monads that we already saw in the guise of IO instructions

Parsing is ...

A functor

Check the functorial laws!

An applicative

Check applicative laws!

A monad

Check the monad laws (upcoming)!

Consequence

Can use `do` notation for parsing!

Parsers are Applicative!

```
instance Applicative (Parser' token) where
  pure = return
  (<*>) = ap

instance Alternative (Parser' token) where
  empty = mzero
  (<|>) = mplus
```

Wrapup

- what if there are multiple applicatives?

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation
- applicatives cannot express dependency

Wrapup

- what if there are multiple applicatives?
- they just compose (unlike monads)
- applicative do notation
- applicatives cannot express dependency
- enable more clever parsers