

\*

# Da the Lamb.

(or at least until there's a vote on the name)

## Lecture 7

# Functors and Applicative Functors

Jeroen Weijers (jeroen.weijers@uni-tuebingen.de)  
<http://db.inf.uni-tuebingen.de/teaching/ws1112/afp>

\* Some people are working on a mascot for Haskell. See the Haskell-cafe thread:  
<http://www.haskell.org/pipermail/haskell-cafe/2011-November/096840.html>

# THE FUNCTOR CLASS

# map

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map f []     = []
```

```
map :: (a -> b) -> Maybe a -> Maybe b
map f (Just a) = Just $ f a
map f Nothing  = Nothing
```

```
map :: (a -> b) -> Either e a -> Either e b
map f (Left e)   = Left e
map f (Right a)  = Right $ f a
```

# The Functor Class

A Functor can be thought of as a container.

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

The Functor class provides functions that can change the values in the container without changing its structure!

A default instance for (<\$) is provided:

```
v <$ c = fmap (const v) c
```

# Examples

*fmap f x* applies function *f* to all elements in “container” *x*

```
> fmap (\x → x + 1) [1,2,3]
[2,3,4]
> fmap (\x → x + 1) (Just 41)
Just 42
> fmap (\x → x + 1) Nothing
Nothing
> fmap (\x → x + 1) (Right 41)
Right 42
> fmap (\x → x + 1) (Left False)
Left False
```

# The Functor Laws

The first functor law is the identity law:

```
fmap id = id
```

```
fmap id s = s
```

If we replace every element of the structure by itself this results in the same structure

# Breaking the Laws

```
instance Functor [] where
  fmap f (x:xs) = map f xs
  fmap f []     = []
```

This instance obviously breaks the identity law:

```
fmap id [1,2,3] == [1,2,3]
=> Definition of (fmap id (1:[2,3]))
map id [2,3] == [1,2,3]
=> Definition of map
[2,3] == [1,2,3]
```

Clearly this is nonsense!

# The Functor Laws

The second functor law is the Composition law:

$$(fmap\ f)\ .\ (fmap\ g)\ =\ fmap\ (f\ .\ g)$$

$$fmap\ f\ \$\ fmap\ g\ x\ =\ fmap\ (f\ .\ g)\ x$$

Mapping a function  $g$  over the elements in a structure and then function  $f$  is equivalent to mapping  $(f . g)$  over a structure

# Breaking the Laws

```
instance Functor [] where
  fmap f (x:xs) = map f xs
  fmap f []     = []
```

This instance obviously breaks the identity law:

```
fmap (+1) $ fmap (+1) [1,2,3]
=> Definition of fmap twice
[5]
```

```
fmap ((+1) . (+1)) [1,2,3]
=> Definition of fmap
[4,5]
```

Clearly this is nonsense!

# Stretching the Container Analogy

```
instance Functor ((→) e) where
  fmap f g = \x → f (g x)
-- fmap :: (a → b) → (e → a) → (e → b)
```

Instead of thinking of a function as a container one might consider a Functor as a computational context.

# Stretching the Container Analogy

```
newtype Parser a = P (String → [(a, String)])
```

```
instance Functor Parser where
```

```
  fmap f p = P $ \inp → [(f v, out)  
                          | (v, out) ← parse p inp ]
```

A Parser is a computational context that takes a string and produces 0 or more results with a rest string.

# Functors

The functions `fmap`, `(<$>)` and `(<$)` apply a function to an argument that is in a container/computational context.

$$f \ \langle \$ \rangle \ a \ :: \ (a \ \rightarrow \ b) \ \rightarrow \ f \ a \ \rightarrow \ f \ b$$

An ordinary function

The result is stored in the same container!

The argument is stored in a container!

# Quiz

Given a parser  $p$  of type *Parser Int*, provide an expression that transform the *Int* into a *String*.

```
> :t p
p :: Parser Int
> let p' = ... p
> :t p'
p' :: Parser String
```

# Quiz

Given a parser  $p$  of type *Parser Int*, provide an expression that transform the *Int* into a *String*.

```
> :t p
p :: Parser Int
> let p' = show <$> p
> :t p'
p' :: Parser String
```

# Quiz

What are the types and the results of the following expressions?

```
> (+1) <$> []  
[] :: [Int]  
> [1] <$ [1,2,3]  
[[1],[1],[1]] :: [[Int]]  
> Just <$> Just "Hello"  
Just (Just "Hello") :: Maybe (Maybe String)  
> (+) <$ [1,2,3]  
[(+),(+),(+)] :: Num a => [a -> a -> a]
```

# APPLICATIVE FUNCTORS

# Function Application in Haskell

In Haskell functions are curried by default.

```
f :: a → b → c
```

We usually say that  $f$  takes two arguments of type  $a$  and  $b$  and returns something of type  $c$

What really happens when we give it its first argument: we get a new function from  $b \rightarrow c$

# Function Application in Haskell

When we have function f:

```
f :: a → b → c
```

The real type is:

```
f :: a → (b → c)
```

And when we apply it to two arguments:

```
f a b
```

What really happens is:

```
(f a) b
```

# Examples

Mapping a function that expects multiple arguments over a container results in a container containing functions.

```
> fmap (+) [1,2,3]
[(1+), (2+), (3+)] :: [Int → Int]
> fmap (+) (Just 41)
Just (41+) :: Maybe (Int → Int)
> fmap (+) Nothing
Nothing :: Maybe (Int → Int)
> fmap (\x y → x + y) (Right 41)
Right (\y → 41 + y) :: Either a (Int → Int)
> fmap (\x y → x + y) (Left False)
Left False :: Either Bool (Int → Int)
```

*GHCi doesn't print functions!*

# Functions in Containers

It is easy to end up with functions in containers, for example:

```
(+) <$> Just 1 :: Maybe (Int → Int)
```

But normal function application doesn't work:

```
> ((+) <$> Just 1) (Just 2)
```

*The function `(+) <\$> Just 1` is applied to one argument, but its type `Maybe (b0 -> a0)` has none*

*In the expression: `((+) <\$> Just 1) (Just 2)`*

# Functions in Containers

We could define an operator that applies a container of functions to a container of arguments:

```
> ((+) <$> Just 1) <*> Just 2  
Just 3
```

```
(<*>) :: Maybe (a → b) → Maybe a → Maybe b  
(Just f) <*> (Just v) = Just $ f v  
_ <*> _ = Nothing
```

Or in terms of fmap:

```
(<*>) :: Maybe (a → b) → Maybe a → Maybe b  
(Just f) <*> v = fmap f v  
Nothing <*> _ = Nothing
```

# Functions in Containers

We can also do that for the Either container:

```
> ((+) <$> Right 1) <*> Right 2  
Right 3
```

```
(<*>) :: Either e (a → b) →  
      Either e a → Either e b  
(Left e) <*> _ = Left e  
(Right f) <*> v = fmap f v
```

# Functions in Containers

What should we do for a list of functions?

- Apply every function to one element in the argument list?
- Apply every function to every element in the argument list?

# Functions in Containers

Apply every function in the list to one element in the argument list:

```
> [(1+), (2*)] <*> [1,2,3]
[2,4]
```

The definition of (`<*>`) now is:

```
(<*>) :: [a → b] → [a] → [b]
fs <*> xs = [f a | v <- zipWith ($) fs xs]
```

# Functions in Containers

Apply every function in the list to every element in the argument list:

```
> [(1+), (2*)] <*> [1,2,3]
[2,3,4,2,4,6]
```

The definition of (`<*>`) now is:

```
(<*>) :: [a → b] → [a] → [b]
fs <*> xs = [f a | f <- fs, a <- xs]
```

This is Haskell's standard implementation for lists!

# The Applicative Functor Class

An applicative functor can be thought of as a container of functions!

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

The applicative functor class provides functions that allow containers of functions to be applied to containers of arguments.

# The Function Pure

The class contains a function called pure, with type:

```
pure :: a → f a
```

This function takes a value and puts it into a container.

Usually this has to be a minimal container!

# The Function Pure

For maybe, pure can be defined as:

```
pure :: a → Maybe a  
pure = Just
```

Or alternatively:

```
pure :: a → Maybe a  
pure x = Just x
```

# The Function Pure

For Either, pure can be defined as:

```
pure :: a → Either e a  
pure = Right
```

Or alternatively:

```
pure :: a → Either e a  
pure x = Right x
```

# The Function Pure

For List, pure can be defined as:

```
pure :: a -> [a]
pure x = [x]
```

# Examples

```
> (+) <$> [1,2,3]
[(1+), (2+), (3+)]
> ((+) <$> [1,2,3]) <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
> fmap (+) (Just 1)
Just (1+)
> fmap (+) (Just 1) <*> Just 2
Just 3
> (pure (+) <*> Just 1) <*> Just 2
Just 3
```

# Associativity and Binding Strength

The operators are declared in Haskell as:

```
infixl 4 <$>  
infixl 4 <*>
```

Both operators are left  
associative and bind  
equally strong

# Associativity and Binding Strength

The operators are declared in Haskell as:

```
infixl 4 <$>  
infixl 4 <*>
```

This means that when we write:

```
f <$> a1 <*> a2 <*> a3
```

Haskell inserts parentheses like:

```
((f <$> a1) <*> a2) <*> a3
```

# Associativity and Binding Strength

Function application still binds stronger!

When we write:

```
pure f <*> a1 <*> g a2 <*> a3
```

Haskell inserts parentheses like:

```
((pure f) <*> a1) <*> (g a2) <*> a3
```

# Quiz

What type does `f` have to be in the following functions?

```
(f <*> Just 5) :: Maybe Int
```

```
f :: Maybe (Int → Int)
```

```
(f <*> Just 5 <*> Just True) :: Maybe Int
```

```
((f <*> Just 5) <*> Just True) :: Maybe Int
```

```
f :: Maybe (Int → Bool → Int)
```

# Some Extra Functions

In the Applicative module the following functions exist:

```
(<*) :: Applicative f => f a -> f b -> f a
```

```
(*>) :: Applicative f => f a -> f b -> f b
```

They are defined as:

```
f <* g = const <$> f <*> g
```

```
f *> g = flip const <$> f <*> g
```

These functions ignore the values of one of their arguments but not the structure!

# Example

```
> Just 1 <*> Just 2
Just 1
> Just 1 <*> Nothing
Nothing
> [1,2] *> [3,4]
[3,4,3,4]
> [] *> [3,4]
[]
```

# Quiz

What type does `f` have to be in the following functions?

```
(f <$> Just 5 <*> Nothing) :: Maybe Int
```

```
f :: Int → Int
```

```
(f <$> Just 5 *> Just True) :: Maybe Bool
```

```
((f <$> Just 5) *> Just True) :: Maybe Bool
```

```
f :: Int → ... -- anything will do for ...
```

# Quiz

What type does  $f$  have to be in the following functions?

```
(f <$> Just 5 <*> Just 2 <*> Just 6)
                                     :: Maybe Int
```

```
((f <$> Just 5) <*> Just 2) <*> Just 6)
                                     :: Maybe Int
```

```
f :: Int → Int → Int
```

# Stretching the Container Analogy

```
newtype Parser a = P (String → [(a, String)])
```

```
instance Applicative Parser where  
  pure v = P $ \inp → [(v, inp)]  
  f <*> g = P $ \inp →  
    concat [parse (v <$> g) out  
            | (v, out) ← parse f inp ]
```

A Parser is a computational context that takes a string and produces 0 or more results with a rest string.

# Examples

```
f <$> natural <*> symbol "+" <*> natural
```

This expressions parses strings like “12 + 30”

Now let's derive the type of  $f$ :

- natural has type: `Parser Int`

```
(f <$> natural) :: Parser ...  
where  
  f :: Int → ...
```

# Examples

```
f <$> natural <*> symbol "+" <*> natural
```

This expressions parses strings like "12 + 30"

Now let's derive the type of *f*:

- symbol has type: `String → Parser String`

```
((f <$> natural) <*> symbol "+") :: Parser ...
```

**where**

```
f :: Int → String → ...
```

# Examples

```
f <$> natural <*> symbol "+" <*> natural
```

This expressions parses a strings like "12 + 30"

Now let's derive the type of  $f$ :

- natural has type: `Parser Int`

```
((f <$> natural) <*> symbol "+") <*> natural)
:: Parser ...
```

**where**

```
f :: Int → String → Int → ...
```

# Examples

```
((f <$> natural) <*> symbol "+") <*> natural
                                :: Parser Int
```

where

```
f :: Int → String → Int → Int
f x _ y = x + y
```

So with this f the following sub-expressions have type:

```
f <$> natural :: Parser (String → Int → Int)
f <$> natural <*> symbol :: Parser (Int → Int)
```

# Examples

Sometimes we want to succeed in parsing something but want to ignore the result value:

We are not interested in the result of symbol “a”

```
f <$> natural <*> symbol “+” <*> natural
where
  f :: Int → String → Int → Int
  f x _ y = x + y
```

# Examples

Sometimes we want to succeed in parsing something but want to ignore the result value:

We can use ( $\langle * \rangle$ ) and ( $\langle * \rangle$ ) to ignore specific results:

```
f <$> natural <* symbol "+" <*> natural
where
  f :: Int → Int → Int
  f = (+)
```

We are interested in the left result (insert parentheses to see why)!

# Quiz

What is the type of  $f$ ?

```
(f <$ symbol "let" <*> identifier  
  <*> symbol "=" <*> expr  
  <*> symbol "in" <*> expr) :: Parser Expr
```

where

```
f :: String → Expr → Expr → Expr
```

```
f = ...
```

# One More Example

We can also “map” over “intermediate” results:

```
> let s = (+) <$> natural <* symbol “+” <*>  
          (product <$> greedy natural)  
> parse s “1 + 1 2 3 4”  
[(25, “”)]
```

# The Applicative Laws

The applicative class has 5 laws.

The most important of these laws is:

```
fmap f x = pure f <*> x
```

These laws hold for most sensible instances! In general it is a good idea to verify that these laws hold when you write a library!

# THE ALTERNATIVE CLASS

# Example

Sometimes a computation might fail and when that happens we want to try an alternative. For example:

```
(+) <$> Just 1 <*> (f <|> g)
  where
    f, g :: Maybe Int
```



If f evaluates to  
Nothing try using g.

# Example

Sometimes a computation might fail and when that happens we want to try an alternative. For example:

```
> (+) <$> Just 1 <*> (Nothing <|> Just 2)
Just 3
> (+) <$> Just 1 <*> (Just 2 <|> Just 3)
Just 3
> (+) <$> Nothing <*> (Just 2 <|> Just 3)
Nothing
> (+) <$> Just 1 <*> (Nothing <|> Nothing)
Nothing
```

# <|> for Maybe a

```
(<|>) :: Maybe a -> Maybe a -> Maybe a  
(Just s) <|> _ = Just s  
Nothing <|> g = g
```

# Example

Lists can be used to encode failure (`[]`) or one or more successes. A list can thus contain all successful alternatives!

```
> (+) <$> [1] <*> ([[] <|> [2,3]]
[3,4])
> (+) <$> [1] <*> ([2,3] <|> [4,5])
[3,4,5,6])
> (+) <$> [] <*> ([2] <|> [3])
[]
> (+) <$> [1] <*> ([[] <|> []])
[]
```

# <|> for [a]

```
(<|>) :: [a] -> [a] -> [a]  
x <|> y = x ++ y
```

Or alternatively:

```
(<|>) :: [a] -> [a] -> [a]  
(<|>) = (++)
```

# The Alternative Class

An Applicative Functor that is an instance of Alternative can be thought of as computations that have a notion of failure!

```
class Applicative f => Alternative f where
  (<|>) :: f a -> f a -> f a
  empty :: f a
```

The Alternative class provides functions that perform a computation and in case of failure tries another computation.

# The Empty Function

The class contains a function called `empty`, with type:

```
empty :: f a
```

This function constructs a container of some type but doesn't take any values to put into the container.

We cannot generate values out of thin air so the container must be empty!

# The Empty Function

For maybe empty can be defined as:

```
empty :: Maybe a  
empty = Nothing
```

# The Empty Function

For [] empty can be defined as:

```
empty :: [a]  
empty = []
```

# The Alternative Laws

Unlike Functor and Applicative Functor, Alternative doesn't have any specific laws.

This doesn't mean that your instances should just do something, something that does make sense is still preferred!

# Associativity and Binding Strength

The operator is declared in Haskell as:

```
infixl 3 <|>
```

This means that we write:

```
f <|> g <|> h <|> i
```

Haskell inserts parentheses like:

```
((f <|> g) <|> h) <|> i
```

First try *f* then *g* then *h*  
and lastly *i*

# Associativity and Binding Strength

Remember that  $\langle * \rangle$  and  $\langle \$ \rangle$  were declared as:

```
infixl 4  $\langle * \rangle$ 
infixl 4  $\langle \$ \rangle$ 
infixl 3  $\langle | \rangle$ 
```

This means that  $\langle * \rangle$  and  $\langle \$ \rangle$  bind stronger! So:

```
f  $\langle \$ \rangle$  g  $\langle | \rangle$  a  $\langle * \rangle$  b  $\langle * \rangle$  c  $\langle | \rangle$  d
```

means:

```
((f  $\langle \$ \rangle$  g)  $\langle | \rangle$  ((a  $\langle * \rangle$  b)  $\langle * \rangle$  c))  $\langle | \rangle$  d
```

# Alternative Parsers

Alternative parser simply try both parsers but prefer the first option over the second!

```
instance Alternative Parser where
  f <|> g = P $ \inp → parse f inp
                    ++ parse g inp
  empty = P $ \inp → []
```

An empty parser is a failing parser (we cannot do anything else, considering the type).

# Examples

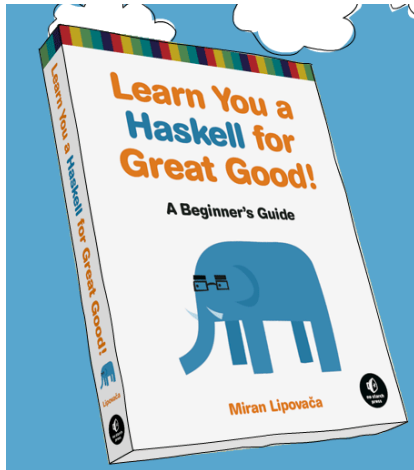
```
> parse (char 'a' <|> char 'b') "ba"  
[('b', "a")]  
> parse ((\x y -> [x,y]) <$> item  
        <*> (char 'a' <|> char 'b')) "ab"  
[("ab", [])]  
> parse (length <$> many (char 'a'))  
        <*> (char 'b' <|> char 'c') "aaaaaaaaac"  
[(8, "")]
```

# Alternative and Applicative Extra's

The module `Control.Applicative` contains both the `Alternative` and `Applicative` classes. In addition it also defines a few useful combinators!

Look them up in the [Library Documentation](#)!

# Reading material



- Learn you a Haskell:
  - Chapter 8, section the Functor type class
  - Chapter 11
  
- The Typeclassopedia
  - Chapter in The Monad Reader issue 13 by Brent Yorgey. Pages 18 - 28