

Monády



Monady sú použiteľný nástroj pre programátora poskytujúci:

- modularitu – skladať zložitejšie výpočty z jednoduchších (no side-effects),
- flexibilitu – výsledný kód je ľahšie adaptovateľný na zmeny,
- izoluje side-effect operácie (napr. IO) od čisto funkcionálneho zvyšku.

Štruktúra prednášok:

- Monády - prvý dotyk
 - Functor
 - Applicative
 - Monády – princípy a zákony
- Najbežnejšie monády
 - Maybe/Error monad
 - List monad
 - IO monad
 - State monad
 - Reader/Writer monad
 - Continuation monad
- Transformátory monád
- Monády v praxi

Monády – úvod



- Phil Wadler: <https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>
Monads for Functional Programming In *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995,
- Noel Winstanley: What the hell are Monads?, 1999
<http://www-users.mat.uni.torun.pl/~fly/materialy/fp/haskell-doc/Monads.html>
- Jeff Newbern's: All About Monads https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf
- A Gentle Introduction to Haskell,
<https://www.haskell.org/tutorial/monads.html>
https://wiki.haskell.org/All_About_Monads
- Sujit Kamthe: Understanding Functor and Monad With a Bag of Peanuts
<https://medium.com/beingprofessional/understanding-functor-and-monad-with-a-bag-of-peanuts-8fa702b3f69e>
- Functors, Applicatives, And Monads In Pictures
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
- Monads in Haskell and Category Theory
<https://www.diva-portal.org/smash/get/diva2:1369286/FULLTEXT01.pdf>



Monads, Arrows, and Idioms

Philip Wadler, <https://homepages.inf.ed.ac.uk/wadler/topics/monads.html>

Články Phila Wadlera na stránke

- Monads for functional programming
- The essence of functional programming
- Comprehending monads
- The arrow calculus
- Monadic constraint programming
- Idioms are oblivious, arrows are meticulous, monads are promiscuous
- The marriage of effects and monads
- How to declare an imperative
- Imperative functional programming

What the hell are Monads?

Noel Winstanley,

<https://www-users.mat.uni.torun.pl/~fly/materialy/fp/haskell-doc/Monads.html>

Obsah:

- Maybe
- State
- The Monad Class
- Do notation
- Monadic IO
- Programming in the IO Monad



All About Monads

Jeff Newbern, https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

Obsah:

Part I - Understanding Monads

What is a monad? Meet the Monads. Doing it with class Monad support in Haskell

Part II - A Catalog of Standard Monads

Introduction. The Identity monad. The Maybe monad. The Error monad. The List monad. The IO monad. The State monad. The Reader monad. The Writer monad. The Continuation monad.

Part III - Monads in the Real

Combining monads the hard way. Monad transformers. Standard monad transformers. Anatomy of a monad transformer. More examples with monad transformers. Managing the transformer.

Roadmap

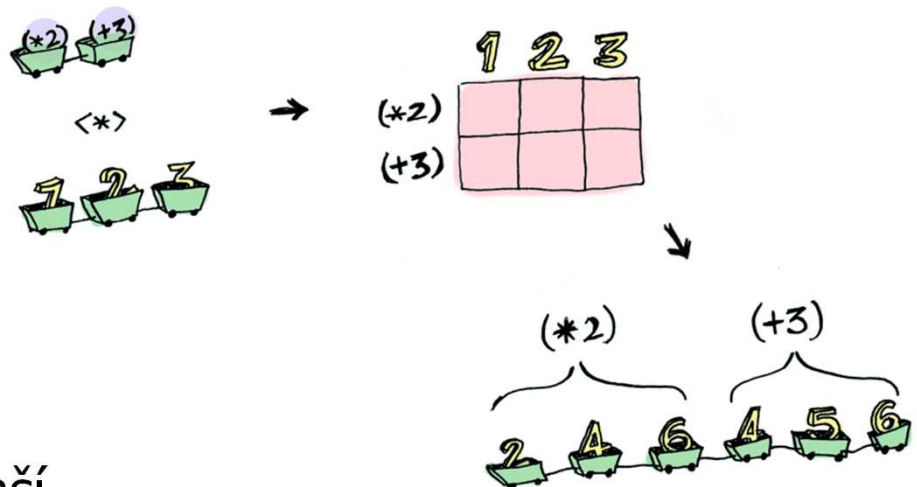
Když egyptský král Ptolemaios žádal slavného matematika Euklida o jednodušší cestu k pochopení matematiky (jako farao se nechtěl obtěžovat těžkou prací studenta), Euklides mu nekompromisně odpověděl: "V matematice neexistuje žádná královská cesta."

- Haskell má triedy, ale sú to vlastne konceptuálne interface (Java)
- Haskell má podtriedy, čo je konceptuálne dedenie na interface (Java)
- dedenie na interface ste určite v Jave videli, napr. na kolekciách

Relevantné triedy v Haskell:

- Functor
- Applicatives
- Monad
- MonadPlus
- ...

Takže monáda nie je najjednoduchší typ v tejto hierarchii



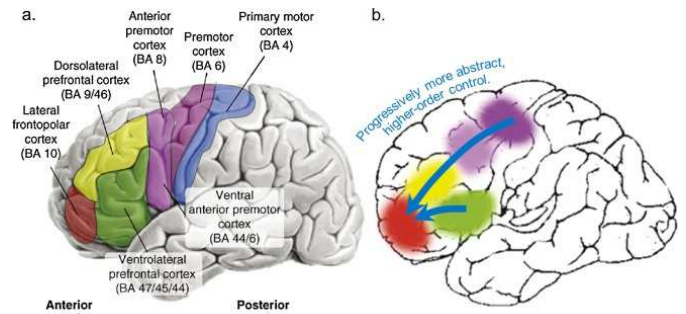
Alternatívny prístup:

Functors, Applicatives, And Monads In Pictures

<http://adit.io/posts/2013-04-17-functors, applicatives, and monads in pictures.html>

Functor

prvotná idea



Development of abstract thinking during childhood and adolescence: The role of rostralateral prefrontal cortex

```
double :: [Int] -> [Int]
double [] = []
double (x:xs) = (x+x):double xs
```

```
sqr :: [Int] -> [Int]
sqr [] = []
sqr (x:xs) = x*x: sqr xs
```

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

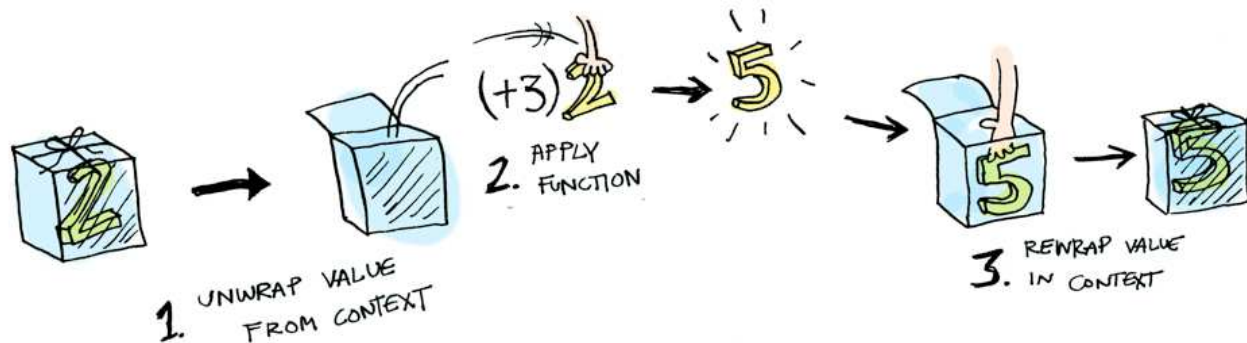
map (*2)

map (^2)

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

fmap aplikuje funkciu f na hodnoty zabalené do typu, ktorý implementuje interface Functor

Functor



<http://adit.io/posts/2013-04-17-functors, applicatives, and monads in pictures.html>

Zoberme jednoduchšiu triedu, z modulu Data.Functor je definovaná takto:

class Functor t where

fmap :: (a -> b) -> t a -> t b

- každý typ t, ak implementuje Functor t,
- musí mať funkciu fmap s profilom
- haskell class je podobne java interface

a každá jej inštancia musí spĺňať dve pravidlá (to je sémantika, mimo syntaxe)

- $\text{fmap id} = \text{id}$ -- identita
- $\text{fmap (p . q)} = (\text{fmap p}) . (\text{fmap q})$ -- kompozícia

Cvičenie1: Príklad inštancie pre **data M₁ a = Raise String | Return a**,
overte, že platia obe sémantické pravidlá:

instance Functor M1 where

fmap **f** (Raise str) = Raise str

fmap **f** (Return x) = Return (**f** x)

Cvičenie

Cvičenie1 (pokrač.):

- $\text{fmap id} =? \text{id}$
 - $\text{fmap id (Raise str)} = \text{Raise str}$
 - $\text{fmap id (Return x)} = \text{Return (id x)} = \text{Return x}$
- $\text{fmap (p.q)} =? (\text{fmap p}) . (\text{fmap q})$
 - Prípad Raise error:
 - L.S. = $\text{fmap (p.q) (Raise str)} = \text{Raise str}$
 - P.S. = $((\text{fmap p}) . (\text{fmap q})) (\text{Raise str}) = (\text{fmap p}) ((\text{fmap q}) (\text{Raise str})) = \text{Raise str}$
 - Prípad Return hodnota:
 - L.S. = $\text{fmap (p.q) (Return x)} = \text{Return ((p.q) x)} = (\text{Return (p (q x))})$
 - P.S. = $((\text{fmap p}) . (\text{fmap q})) (\text{Return x})$
= $(\text{fmap p}) ((\text{fmap q}) (\text{Return x}))$
= $(\text{fmap p}) (\text{Return (q x)}) = (\text{Return (p (q x))}) \dots \text{q.e.d.}$

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

Definícia:

```
fmap f (Raise str)    = Raise str  
fmap f (Return x)    = Return (f x)
```

Dokázať:

```
fmap id              = id  
fmap (p . q)         = (fmap p) . (fmap q)
```

Functor

Maybe, List

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

fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

Cvičenie2: Definujte inštanciu triedy Functor pre typy:

data MyMaybe a = MyJust a | MyNothing deriving (Show) -- alias Maybe a

data MyList a = Null | Cons a (MyList a) deriving (Show) -- alias [a]

... a pochopíte, ako je Functor definovaný pre štandardné typy Maybe a [].

```
> fmap (even) (Cons 1 (Cons 2 Null))           -- f : Int->Bool
```

```
Cons False (Cons True Null)
```

```
> fmap (\s -> s+s) (Cons 1 (Cons 2 Null))       -- f : Int->Int
```

```
Cons 2 (Cons 4 Null)
```

```
> fmap (show) (Cons 1 (Cons 2 Null))            -- f : Int->String
```

```
Cons "1" (Cons "2" Null)
```

```
> fmap ((\t -> t++t) . (show)) (Cons 1 (Cons 2 Null)) -- f : (String->String).(Int->String)
```

```
Cons "11" (Cons "22" Null)
```

```
> fmap (\t -> t++t) (fmap (show) (Cons 1 (Cons 2 Null))) -- "overenie" vlastnosti kompozície
```

```
Cons "11" (Cons "22" Null)
```

```
> fmap id (Cons 1 (Cons 2 Null))                -- overenie vlastnosti identity
```

```
Cons 1 (Cons 2 Null)
```

Functor

Maybe, List

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

fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

Cvičenie2 (pokrač.): Definujte inštanciu triedy Functor pre typy:

data MyMaybe a = MyJust a | MyNothing deriving (Show) -- alias Maybe a

data MyList a = Null | Cons a (MyList a) deriving (Show) -- alias [a]

```
instance Functor MyMaybe where
  fmap f MyNothing  = MyNothing
  fmap f (MyJust x) = MyJust (f x)
```

```
instance Functor MyList where
  fmap f Null = Null
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

```
instance Functor [] where
  fmap = map
```

... stále ale chýba dôkaz platnosti dvoch vlastností ...

```
> fmap even [1,2,3]
[False,True,False]
> fmap (*2) [1,2,3]
[2,4,6]
> fmap (show) [1,2,3]
["1","2","3"]
> fmap (\x->x++x) $ fmap (show) [1,2,3]
["11","22","33"]
> fmap ((\x->x++x). show) [1,2,3]
["11","22","33"]
```

Functor – strom

```
class Functor t where
  fmap :: (a -> b) -> t a -> t b
  fmap id      = id
  fmap (p . q) = (fmap p) . (fmap q)
```

Cvičenie3: Binárny strom (skoro ako tradičný LExp, ale parametrizovaný typ):
data LExp a = ID a | APP (LExp a) (LExp a) | ABS a (LExp a) deriving (Show)

instance Functor LExp where

fmap f (ID x)	= ID (f x)
fmap f (APP left right)	= APP (fmap f left) (fmap f right)
fmap f (Abs x body)	= ABS (f x) (fmap f body)

```
omega = ABS "x" (APP (ID "x") (ID "x"))
> fmap (\t -> t++t) omega
ABS "xx" (APP (ID "xx") (ID "xx"))
> fmap (\t -> (length t)) omega
ABS 1 (APP (ID 1) (ID 1))
```

Cvičenie4: Ľubovoľne n-árny strom (prezývaný RoseTree alias Rhododendron):
data RoseTree a = Rose a [RoseTree a]

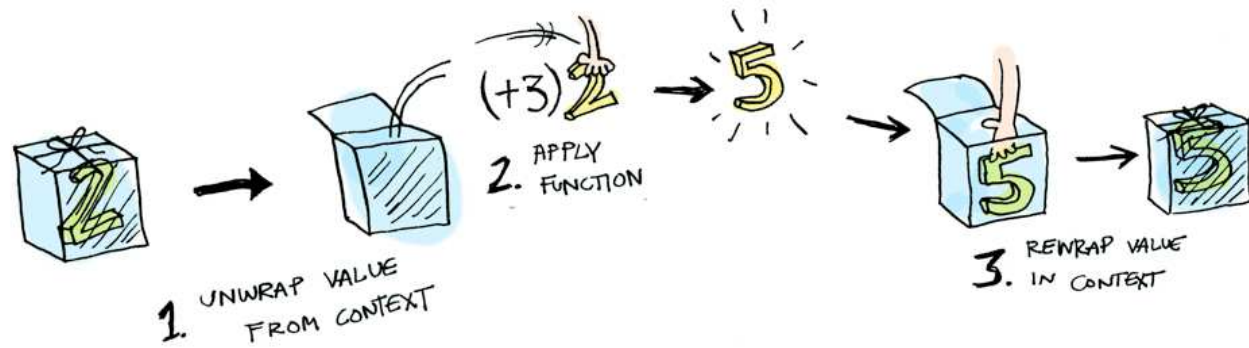
instance Functor RoseTree where

fmap **f** (Rose a bs) = Rose (**f** a) (map (fmap **f**) bs)

... opäť chýba dôkaz platnosti vlastností pre Cvičenie 3 aj 4 ...

Functor

zhrnutie



<http://adit.io/posts/2013-04-17-functors, applicatives, and monads in pictures.html>

```
instance Functor [] where
  -- fmap :: (a->b) -> [a] -> [b]
  fmap = map
```

```
instance Functor Maybe where
  -- fmap :: (a->b) -> Maybe a -> Maybe b
  fmap _ Nothing = Nothing
  fmap g (Just x) = Just (g x)
```

```
instance Functor IO where
  -- fmap :: (a->b) -> IO a -> IO b
  fmap g mx = do { x<-mx; return (g x) }
```

infixl 4 <\$>

```
<$> = fmap
main :: IO ()
main = do res <- words <$> getLine
          res <- fmap words getLine
          putStrLn $ show res
```

```
double :: Functor f => f Int -> f Int
double = fmap (*2)
```

```
sqr :: Functor f => f Int -> f Int
sqr = fmap (^2)
```

```
double (Just 7) = Just 14
double [1,2,3,4] = [2,4,6,8]
```

```
double (Branch (Leaf 7) (Leaf 9)) =
  Branch (Leaf 14) (Leaf 18)
```

```
double (Rose 3 [Rose 5 [], Rose 7 []]) =
  Rose 6 [Rose 10 [], Rose 14 []]
```

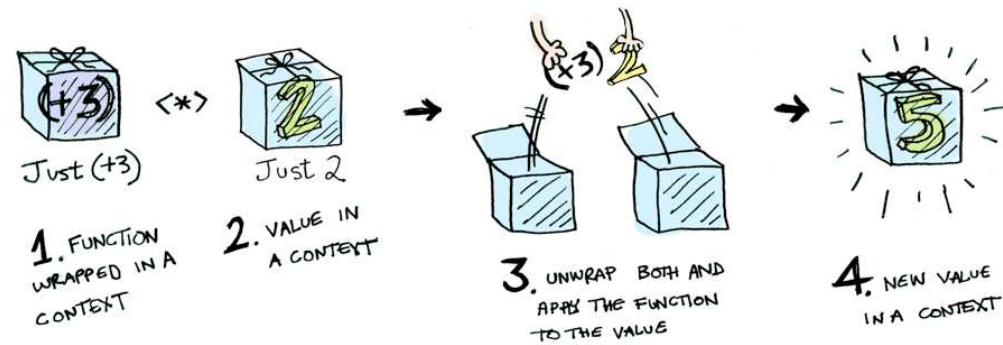


Applicative

prvotná idea

- Functor predstavuje abstrakciu aplikácie **unárnej funkcie** na každý prvok “Functor-like” dátovej štruktúry, nech je akákoľvek komplikovaná...
- Čo, ak by sme mali funkcie s veľa argumentami (nie len unárne):
 - `fmap0 :: a -> f a`
 - `fmap1 :: (a->b) -> f a -> f b` -- fmap
 - `fmap2 :: (a->b->c) -> f a -> f b -> f c`
 - `fmap3 :: (a->b->c->d) -> f a -> f b -> f c -> f d` -- ☹
- riešenie = **Currying** je transformácia funkcie s mnohými argumentami na unárnu, ktorá vráti inú funkciu, ktorá skonzumuje všetky ďalšie argumenty
 - **pure** `:: a -> f a`
 - **(<*>)** `:: f (a->b) -> f a -> f b` **infixl**
Např. nech g chce “tri argumenty”
 - `pure g <*> x <*> y <*> z = ((pure g <*> x) <*> y) <*> z`
- Hierarchia -- x :: f a, y :: f b, z :: f c
 - `pmap0 g0 = pure g0` -- g0 je konštanta, lebo má 0-args.
 - `fmap1 g1 x = pure g1 <*> x` -- g1 :: a->b, pure g1 :: f (a->b)
 - `fmap2 g2 x y = pure g2 <*> x <*> y` -- g2 :: a->b->c, pure g2::f (a->b->c)
 - `fmap3 g3 x y z = pure g3 <*> x <*> y <*> z` -- ...

Applicative



<http://adit.io/posts/2013-04-17-functors, applicatives, and monads in pictures.html>

class **Functor** **f** => **Applicative** **f** where -- Applicative je podtrieda Functor

pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b (infixl 4)

a každá jej inštancia musí spĺňať pravidlá (to je sémantika, mimo syntaxe)

- pure id <*> v = v -- identita
- pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia
- pure f <*> pure x = pure (f x) -- homomorfizmus
- u <*> pure y = pure (\$ y) <*> u = pure (\g->g y) <*> u -- výmena

Príklad (pre M1):

Return id <*> Return 4 = Return 4

Return (.) <*> Return (+1) <*> Return (+2) <*> Return 4 = Return 7

Return (+1) <*> (Return (+2) <*> Return 4) = Return 7

Return (+4) <*> Return 3 = Return 7 -- pure f <*> pure x = pure (f x)

data M1 a = Raise String | Return a deriving(Show, Read, Eq)

Applicative

class **Functor** f => Applicative f where

pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v

-- identita

pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia

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

-- homomorfizmus

u <*> pure y = pure (\$ y) <*> u = pure (\g->g y) <*> u

Cvičenie5: definujte inštanciu M1 pre triedu Applicative a overte 4 pravidielá:
instance Applicative M1 where

pure a = Return a

(Raise e) <*> _ = Raise e

-- e:: String, Raise e::M1 a

(Return f) <*> a = fmap f a

-- f::a->b, Return f :: M1(a->b)

Príklad:

1) Return id <*> Return 4 = Return 4

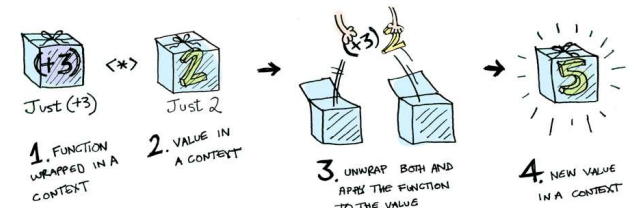
2) L.S. = Return (.) <*> Return (+1) <*> Return (+2) <*> Return 4 = Return 7

P.S. = Return (+1) <*> (Return (+2) <*> Return 4) = Return 7

3) Return (+4) <*> Return 3 = Return 7

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

4) Return (+2) <*> Return 7 = Return 9 = Return (\$ 7) <*> Return (+2)



data M1 a = Raise String | Return a deriving(Show, Read, Eq)

Applicative

```
class Functor f => Applicative f where
```

```
  pure :: a -> f a
```

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

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

-- identita

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

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

-- homomorfizmus

```
  u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie5 (pokrač.): definujte inštanciu M1 pre Applicative a overte pravidlá:
instance Applicative M1 where

pure a = Return a

(Raise e) <*> _ = Raise e

-- e:: String, Raise e::M1 a

(Return f) <*> a = fmap f a

-- f::a->b, Return f :: M1(a->b)

Dôkaz:

1) (Return id) <*> v = fmap id v = v

pravidlo identity pre Functors

3) pure f <*> pure x = (Return f) <*> (Return x) = fmap f (Return x) = Return (f x) = pure (f x)

2) (Return (.)) <*> (Return fu) <*> (Return fv) <*> (Return fw) =

(Return (.) fu) <*> (Return fv) <*> (Return fw) =

(Return ((.) fu) fv) <*> (Return fw) = (Return (fu . fv)) <*> (Return fw) =

(Return ((fu . fv) fw)) = Return (fu (fv (fw)))

4) L.S. = (Return f) <*> (Return y) = fmap f (Return y) = (Return (f y))

P.S. = (Return (\$ y)) <*> (Return f) = fmap (\$ y) (Return f) = Return ((\$ y) f) =

Return ((\g->g y) f) = Return (f y)

```
data M1 a = Raise String | Return a deriving(Show, Read, Eq)
```

Applicative

class **Functor** f => Applicative f where

pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v

-- identita

pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia

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

-- homomorfizmus

u <*> pure y = pure (\$ y) <*> u = pure (\g->g y) <*> u

Cvičenie5'': definujte inštanciu Maybe pre triedu Applicative a overte pravidlá:
instance Applicative Maybe where

pure :: a -> Maybe a

pure = Just

pure x = Just x

(<*>) :: Maybe (a->b) -> Maybe a -> Maybe b

Nothing <*> _ = Nothing

(Just g) <*> a = fmap g a

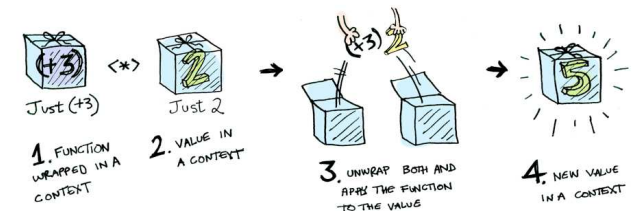
Príklad:

pure (*2) <*> Just 7 = Just 14

pure (+) <*> Just 7 <*> Just 9 = Just 16

pure (+) <*> Nothing <*> Just 9 = Nothing

pure (\x y z->(x,y,z)) <*> Just 1 <*> Just 2 <*> Just 3 = Just (1,2,3)



data Maybe a = Just a | Nothing deriving(Show, Read, Eq)

Applicative

```
class Functor f => Applicative f where
```

```
  pure :: a -> f a
```

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

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

-- identita

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

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

-- homomorfizmus

```
  u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie6: definujte inštanciu [] pre triedu Applicative a overte pravidlá:
instance Applicative [] where

```
  pure a          = [a]
```

```
  fs <*> xs       = [ f x | f <- fs, x <- xs]
```

Príklad:

- 1) pure (+1) <*> [1..5] = [2,3,4,5,6]
 pure (+) <*> [1,2] <*> [11,12] = [12,13,13,14]
 pure (,) <*> [1,2] <*> [11,12] = [(1,11),(1,12),(2,11),(2,12)]
- 2) pure (.) <*> pure (+1) <*> pure (+3) <*> pure 9 = 13
 pure (.) <*> pure (+1) <*> pure (+3) <*> [9] = [13]
 pure (.) <*> pure (+1) <*> pure (+3) <*> Just 9 = Just 13
- 3) pure (+1) <*> pure 7 = 8
- 4) pure (\$ 7) <*> pure (+1) = 8

```
type [a] = [] / a:[a]
```

Applicative

class **Functor** f => Applicative f where

pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v

-- identita

pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia

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

-- homomorfizmu

<*> pure y = pure (\$ y) <*> u = pure (\g->g y) <*> u

Cvičenie6: definujte inštanciu [] pre Applicative, a overte pravidlá:

instance Applicative [] where

pure a = [a]

fs <*> xs = [f x | f <- fs, x <- xs]

Dôkaz:

1) (Return id) <*> v = [id] <*> v = v

2) [(.)] <*> [ui] <*> [vj] <*> [wk] =

[(.)ui] <*> [vj] <*> [wk] =

[ui . vj] <*> [wk] = [(ui . vj) wk] =

[(ui (vj wk))]

3) pure f <*> pure x = [f] <*> [x] = [f x]

4) [f1,... fn] <*> [y] = [f1 y, ... fn y]

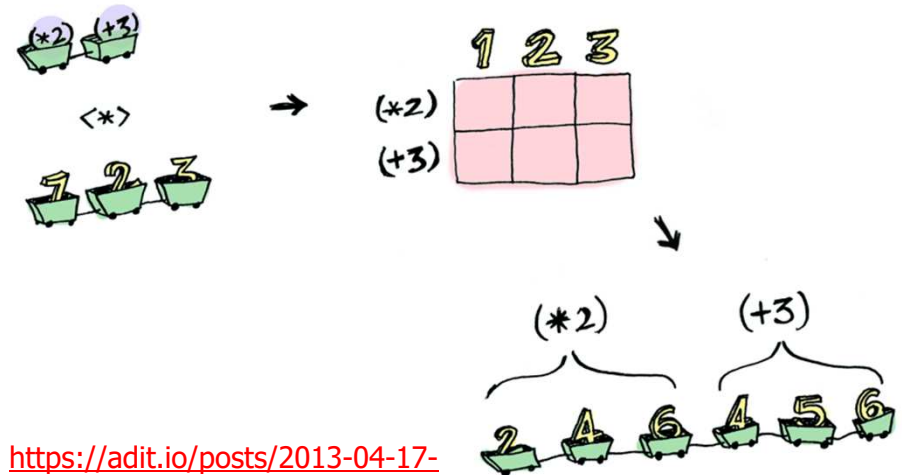
Príklady:

[(*2), (+3)] <*> [1,2,3] = [2,4,6,4,5,6]

(,) <\$> [1,2,3] <*> [4,5,6] = [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

(\x y z -> (x,y,z)) <\$> [1,2] <*> [3,4] <*> [5,6] = [(1,3,5),(1,3,6),(1,4,5),(1,4,6),(2,3,5),(2,3,6),(2,4,5),(2,4,6)]

pure (,,) <*> [1,2] <*> [3,4] <*> [5,6] = [(1,3,5),(1,3,6),(1,4,5),(1,4,6),(2,3,5),(2,3,6),(2,4,5),(2,4,6)]



<https://adit.io/posts/2013-04-17-functors, applicatives, and monads in pictures.html>



Kartézsky súčin

domáca úloha

module KSucin where
cart :: [[t]] -> [[t]]

Príklad:

cart [[1,2], [3,4], [5]] = [[1,3,5],[1,4,5],[2,3,5],[2,4,5]]

cart [[1,2], [3,4], [5,6,7]] = [[1,3,5],[1,3,6],[1,3,7],[1,4,5],[1,4,6],[1,4,7],[2,3,5],[2,3,6],[2,3,7],[2,4,5],[2,4,6],[2,4,7]]

cart [[1,2], [3,4], [5,6]] = [[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]

cart [[1,2], [3,4], []] = []

cart [["janka", "danka"], ["misko", "palko"]] = [["janka","misko"],["janka","palko"],["danka","misko"],
["danka","palko"]]



GHC.Base

- <https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-Base.html>

▸ `Applicative []` # *Since: base-2.1*

▾ `Applicative Maybe` # *Since: base-2.1*

Defined in `GHC.Base`

Methods

```
pure :: a -> Maybe a
```

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
liftA2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

```
(>*) :: Maybe a -> Maybe b -> Maybe b
```

```
(<*) :: Maybe a -> Maybe b -> Maybe a
```

▸ `Applicative IO` # *Since: base-2.1*

Monáda

(class Monad)



monáda **m** je typ implementujúci dve funkcie:

class Applicative m => Monad **m** where

return :: a -> m a

>>= :: m a -> (a -> m b) -> m b

-- interface predpisuje tieto funkcie

-- to bude pure z Applicatives

-- náš `bind`

ktoré splňajú isté (sémantické) zákony:

neutrálnosť return:

- $\text{return } c \gg= (\backslash x \rightarrow g)$ $=$ $g[x/c]$
- $m \gg= \backslash x \rightarrow \text{return } x$ $=$ m

neutrálnosť asociativita:

- $m1 \gg= (\backslash x \rightarrow m2 \gg= (\backslash y \rightarrow m3)) = (m1 \gg= (\backslash x \rightarrow m2)) \gg= (\backslash y \rightarrow m3)$

inak zapísané:

- | | | | |
|---------------------------|-----|--|--------------------------|
| $\text{return } c \gg= f$ | $=$ | $f \ c$ | -- ľavo neutrálny prvok |
| $m \gg= \text{return}$ | $=$ | m | -- pravo neutrálny prvok |
| $(m \gg= f) \gg= g$ | $=$ | $m \gg= (\backslash x \rightarrow f \ x \gg= g)$ | -- asociativita >>= |