

Pokročilá syntaxe, datové struktury, funktory

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2022

Organisace předmětu, domácí úkoly, podmínky zápočtu:

- vizte **interaktivní osnovu** v ISu
- první miniúkol je již zveřejněn (klidně si ho otevřete)
 - technické i jiné problémy hlase do fóra nebo na konci cvičení

Softwarové vybavení: překladač GHC + interpret GHCi

- ve versi alespoň 8.4¹
- na fakultních počítačích: `$ module add ghc`
- na vlastních počítačích: to zvládnete :-) (ale zkusíme vám pomoci)

¹oproti v. 9.2 na FI může být potřeba zapínat některá rozšíření explicitně

Pokročilejší syntaxe: `case`

Připomenutí: `case`: používání vzorů uvnitř funkce

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

```
mapMaybe _ [] = []
```

```
mapMaybe f (x:xs) = case f x of
```

```
    Just v -> v : mapMaybe f xs
```

```
    Nothing -> mapMaybe f xs
```

Pokročilejší syntaxe: `case`

Připomenutí: `case`: používání vzorů uvnitř funkce

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []      = []
mapMaybe f (x:xs) = case f x of
                        Just v  -> v : mapMaybe f xs
                        Nothing ->   mapMaybe f xs
```

Rozšíření LambdaCase pro zkrácení `\x -> case x of`

```
{-# LANGUAGE LambdaCase #-} -- úplně na vrchu souboru
                             -- nebo ghci -XLambdaCase
defaultedMap :: (a -> b) -> b -> [Maybe a] -> [b]
defaultedMap f d = map $ \case
                        Just v  -> f v
                        Nothing -> d
```

Pokročilejší syntaxe: strážce (guards)

Připomenutí: vyhnutí se násobným `if`ům

```
elemBST :: Ord k => k -> BinTree k -> Bool
```

```
elemBST _ Empty = False
```

```
elemBST x (Node v l r)
```

```
  | x == v      = True
```

```
  | x < v       = elemBST x l
```

```
  | otherwise   = elemBST x r
```

Pokročilejší syntaxe: stráže (guards)

Připomenutí: vyhnutí se násobným `if`ům

```
elemBST :: Ord k => k -> BinTree k -> Bool
elemBST _ Empty = False
elemBST x (Node v l r)
  | x == v      = True
  | x < v       = elemBST x l
  | otherwise   = elemBST x r
```

Výchozí rozšíření PatternGuards umožňuje použít ve strážích i vzory

```
-- lookup :: Eq a => a -> [(a, b)] -> Maybe b
defaultedLookup :: [(Int, a)] -> a -> Int -> a
defaultedLookup db def key
  | key >= 0, Just val <- lookup key db = val
  | otherwise                           = def
```

Pokročilejší syntaxe: \$

Připomenutí: zbavujeme se závorek dolarem

$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

Operátor aplikace funkce (priorita 0, asociativita zprava)

■ $f \$ g \$ x \equiv f (g x)$

■ $f . g \$ h x \equiv (f . g) (h x)$

Pokročilejší syntaxe: \$

Připomenutí: zbavujeme se závorek dolarem

$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

Operátor aplikace funkce (priorita 0, asociativita zprava)

■ $f \$ g \$ x \equiv f (g x)$

■ $f . g \$ h x \equiv (f . g) (h x)$

Zamyšlení:

mezera jako operátor aplikace (priorita 10, asociativita zleva)

■ $f g x \equiv (f g) x$

Pokročilejší syntaxe: \$

Připomenutí: zbavujeme se závorek dolarem

`($) :: (a -> b) -> a -> b`

`f $ x = f x`

Operátor aplikace funkce (priorita 0, asociativita zprava)

■ `f $ g $ x ≡ f (g x)`

■ `f . g $ h x ≡ (f . g) (h x)`

Zamyšlení:

mezera jako operátor aplikace (priorita 10, asociativita zleva)

■ `f g x ≡ (f g) x`

```
filter (>10) ( map (~2) ( filter even [1..10] ) )
```

Pokročilejší syntaxe: \$

Připomenutí: zbavujeme se závorek dolarem

`($) :: (a -> b) -> a -> b`

`f $ x = f x`

Operátor aplikace funkce (priorita 0, asociativita zprava)

■ `f $ g $ x ≡ f (g x)`

■ `f . g $ h x ≡ (f . g) (h x)`

Zamyšlení:

mezera jako operátor aplikace (priorita 10, asociativita zleva)

■ `f g x ≡ (f g) x`

```
filter (>10) ( map (^2) ( filter even [1..10] ) )
( filter (>10) . map (^2) ) ( filter even [1..10] )
```

Pokročilejší syntaxe: \$

Připomenutí: zbavujeme se závorek dolarem

`($) :: (a -> b) -> a -> b`

`f $ x = f x`

Operátor aplikace funkce (priorita 0, asociativita zprava)

■ `f $ g $ x ≡ f (g x)`

■ `f . g $ h x ≡ (f . g) (h x)`

Zamyšlení:

mezera jako operátor aplikace (priorita 10, asociativita zleva)

■ `f g x ≡ (f g) x`

```
filter (>10) ( map (^2) ( filter even [1..10] ) )
( filter (>10) . map (^2) ) ( filter even [1..10] )
filter (>10) . map (^2) $ filter even [1..10]
```

Pokročilejší syntaxe: @

Někdy chceme u vstupního argumentu funkce použít vzory, ale zároveň pak použít argument jako celek.

```
foo ((Person sex age weight) : ps)
  = if {- ... -} then (Person sex age weight) : foo ps
    else foo ps
```

Pokročilejší syntaxe: @

Někdy chceme u vstupního argumentu funkce použít vzory, ale zároveň pak použít argument jako celek.

```
foo ((Person sex age weight) : ps)
  = if {- ... -} then (Person sex age weight) : foo ps
      else foo ps
```

Můžeme použít *as-patterns*:

```
foo (p@(Person sex age weight) : ps)
  = if {- ... -} then p : foo ps
      else foo ps
```

Vlastní datové typy: záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
getRight (Node _ _ r) = r
```

Vlastní datové typy: záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
getRight (Node _ _ r) = r
```

Záznamy (*records*): pojmenování hodnot v definici

```
data BinTree a = Empty | Node { btValue :: a
                                , btLeft  :: BinTree a
                                , btRight :: BinTree a
                                }
```

- lze použít pattern-matching jako v prvním případě

Vlastní datové typy: záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
getRight (Node _ _ r) = r
```

Záznamy (*records*): pojmenování hodnot v definici

```
data BinTree a = Empty | Node { btValue :: a
                               , btLeft  :: BinTree a
                               , btRight :: BinTree a
                               }
```

- lze použít pattern-matching jako v prvním případě
- názvy hodnot ve vzorech

```
getRight (Node { btRight = r } ) = r
```


Vlastní datové typy: záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
getRight (Node _ _ r) = r
```

Záznamy (*records*): pojmenování hodnot v definici

```
data BinTree a = Empty | Node { btValue :: a
                                , btLeft  :: BinTree a
                                , btRight :: BinTree a
                                }
```

- lze použít pattern-matching jako v prvním případě

- názvy hodnot ve vzorech

```
getRight (Node { btRight = r } ) = r
```

- názvy hodnot jako funkce

```
getRight node = btRight node
```

Vlastní datové typy: záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
getRight (Node _ _ r) = r
```

Záznamy (*records*): pojmenování hodnot v definici

```
data BinTree a = Empty | Node { btValue :: a
                                , btLeft  :: BinTree a
                                , btRight :: BinTree a
                                }
```

- lze použít pattern-matching jako v prvním případě

- názvy hodnot ve vzorech

```
getRight (Node { btRight = r } ) = r
```

- názvy hodnot jako funkce

```
getRight node = btRight node
```

- modifikace podmnožiny atributů

```
setVal node = node { btValue = 2 }
```

type a newtype

```
type String = [Char]
type Matrix a = [[a]]
```

- typový alias: jen nové pojmenování, zaměnitelné s původním typem
- výjimka: nelze použít při instanciování typových tříd
- pouze pro přehlednost kódu, pro překladač transparentní

type a newtype

```
type String = [Char]
type Matrix a = [[a]]
```

- typový alias: jen nové pojmenování, zaměnitelné s původním typem
- výjimka: nelze použít při instanciování typových tříd
- pouze pro přehlednost kódu, pro překladač transparentní

```
newtype Matrix a = M { unM :: [[a]] } deriving Show
```

- nový typ (jako **data**) \Rightarrow typová kontrola překladače
- musí mít právě jeden unární hodnotový konstruktor
- nutnost „rozbalování“ a „balení“

```
M (map (map (+4)) (unM matrix))
```

- časté využití: psaní jiné instance pro týž typ
- rychlejší než **data**, další rozdíly s rozšířeními GHC (nad rámec kurzu: `-XGeneralisedNewtypeDeriving`)

Typované díry I.

Otypování samostatného výrazu:

- přes povel interpretu :t

Typované díry I.

Otypování samostatného výrazu:

- přes povel interpretu :t

Otypování výrazu v kódu:

- použijeme typovanou díru: ["a", "b", _]
- vygeneruje typovou chybu obsahující požadovaný typ

```
> [ "a", "b", _ ]
```

```
<interactive>:1:12: error:
```

```
  * Found hole: _ :: [Char]
```

```
  * In the expression: ["a", "b", _]
```

```
  ...
```

Typované díry II.

Typované díry:

- názvy začínají podtržítkem
- ladění typových chyb ve složitějším kódu
- prototypování programu (namísto `undefined`)
- chyby lze odložit až do okamžiku volání pomocí přepínače `GHC/GHCi -fdefer-typed-holes`
- pozor: i stejně pojmenované díry jsou *různé* díry
- více v [dokumentaci GHC](#) a na [GHC wiki](#)

Moduly a balíky

Moduly (Data.Char, Data.Map.Lazy, ...)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- modul nemusí exportovat všechny své funkce

```
module Jmeno.Modulu ( FooT( Foo )  
                    , BarT(..)  
                    , fooToBar ) where
```


Moduly a balíky

Moduly (`Data.Char`, `Data.Map.Lazy`, ...)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- modul nemusí exportovat všechny své funkce

```
module Jmeno.Modulu ( FooT( Foo )
                    , BarT(..)
                    , fooToBar ) where
```

Balíky (`containers-0.5.10.1`, `brainfuck-0.1.0.3`, ...)

- skupina modulů, která se instaluje spolu
- název většinou malými písmeny, nemá hierarchii, má verzi
- balík `base` (základní balík modulů)
- spravuje typický balíčkovací systém `cabal`

Informace o modulech/balících:

- databáze balíků **Hackage**
 - resp. **Stackage** („Stable Hackage“)
- vyhledávač **Hoogle** (hledání podle funkcí, typů, balíků, ...)
 - fakultní instance: <https://hoogle.fi.muni.cz>

Informace o modulech/balících:

- databáze balíků **Hackage**
 - resp. **Stackage** („Stable Hackage“)
- vyhledávač **Hoogle** (hledání podle funkcí, typů, balíků, ...)
 - fakultní instance: <https://hoogle.fi.muni.cz>

Použití při programování

- nainstalovat balík přes cabal:
`cabal new-update && cabal new-install --lib balík`
- import v kódu: `import Modul`
- přidání modulu v GHCi: přes `import` nebo `:m + Modul`

Nástroj pro správu balíčků

- aktualizace databáze balíčků: `cabal new-update`
- instalace z **Hackage**: `cabal new-install [--lib] balík`
 - Pozor, `cabal new-uninstall` neexistuje!
- skoro všechno ukládá do `$HOME/.cabal/`
 - do `$PATH` je třeba přidat `$HOME/.cabal/bin`
 - konfiguraci a instalace je možno snadno smazat
 - při mazání je také nutno pročistit adresář `$HOME/.ghc`

Pozor:

- jsme uprostřed výrazné změny fungování
- příkazy `v1-/old-` vs. `v2-/new-`
- příkazy bez prefixu různé u různých verzí

★ Sandboxing

- je nutno založit projekt: `cabal init`
- v souboru `projekt.cabal` nastavit požadované balíky
- `cabal new-build` stáhne a sestaví jen to, co je potřeba
 - různé projekty sdílí stejné balíky
 - je možno mít více verzí jednoho balíku
- GHCi je možné spustit běžně nebo přes `cabal new-repl`
- „sandboxování“ binárek = symlinkování různých verzí
 - není nutné zakládat projekt, jen vybrat umístění odkazů
 - volba `--symlink-bindir=adresář`

Více v [dokumentaci Cabalu](#)

- součástí modulu `ghc`
- na nymfách lokálně nainstalována zastaralá verze, nutno přebít modulem
- na aise je zapotřebí novější překladač `gcc` (`module add gcc`)
- kvůli rozdílné verzi systémových knihoven na aise a nymfách:
 - balíky sestavené na nymfách nebudou fungovat na aise
 - balíky sestavené na aise *by měly* fungovat na nymfách
 - **doporučení: instalaci všech balíčků provádět na aise**

Užitečné datové typy

Datové typy **Set** a **Map**

Množiny a asociativní mapy (též *slovníky*):

- **Set** a
 - **a** je typ hodnoty, musí být v **Ord**
- **Map** **k** **v**
 - **k** je typ klíče, musí být **Ord**
 - **v** je typ hodnoty
- moduly **Data.Set** a **Data.Map**
 - balík `containers`, součást běžné distribuce GHC
 - **Map** dvou typů (*lazy* a *strict*)
 - vhodný kvalifikovaný import
- logaritmické vkládání, odstraňování, zjišťování minima a maxima

Kvalifikovaný import: ukázka

- Selektivní import

```
import Data.Set (Set, empty, insert)
```

```
courses :: Set String
```

```
courses = insert "IB016" empty
```

Kvalifikovaný import: ukázka

- Selektivní import

```
import Data.Set (Set, empty, insert)
```

```
courses :: Set String
```

```
courses = insert "IB016" empty
```

- Kvalifikovaný import

```
import qualified Data.Set as S
```

```
courses :: S.Set String
```

```
courses = S.insert "IB016" S.empty
```

Typový konstruktorek `Either`

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- používá se, když může mít výpočet dva typy výsledků
- často se používá jako zobecnění `Maybe`
 - `Left` `a` označuje chybný výpočet, hodnota specifikuje chybu
 - `Right` `b` označuje korektní výpočet, hodnota je výsledkem

```
myDiv :: Int -> Int -> Either String Int
myDiv x 0 | x < 0 = Left "-inf"
          | x > 0 = Left "+inf"
          | x == 0 = Left "NaN"
myDiv x y = Right (div x y)
```

Druhy (*kinds*)

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu *

```
Integer :: *
```

```
Maybe Int :: *
```

```
Either String Int :: *
```

```
BinTree (Int, [Int]) :: *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu *

```
Integer :: *  
Maybe Int :: *  
Either String Int :: *  
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
[] :: * -> *  
Maybe :: * -> *  
(,) :: * -> * -> *  
Either :: * -> * -> *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu *

```
Integer :: *  
Maybe Int :: *  
Either String Int :: *  
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
[] :: * -> *  
Maybe :: * -> *  
(,) :: * -> * -> *  
Either :: * -> * -> *
```

- opět platí princip částečné aplikace

```
Either String :: * -> *
```

- GHCi definuje povel `:k` na určení druhu

Funktory

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]
```

```
map :: (a -> b) -> [a] -> [b]
```

```
map _ [] = []
```

```
map f (x:xs) = f x : map f xs
```

Motivace: map

- Funkce `map` na seznamech:

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

- Funkce `map` na binárních stromech:

```
data BinTree a = Node a (BinTree a) (BinTree
  ↪ a)
                | Empty
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ Empty = Empty
treeMap f (Node v l r) =
  Node (f v) (treeMap f l) (treeMap f r)
```

Motivace: map

- Funkce `map` na seznamech:

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

- Funkce `map` na binárních stromech:

```
data BinTree a = Node a (BinTree a) (BinTree
  ↪ a)
                | Empty
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ Empty = Empty
treeMap f (Node v l r) =
    Node (f v) (treeMap f l) (treeMap f r)
```

- Nedalo by se to zobecnit? Jaký je obecný typ funkce `map`?

Typová třída Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu $* \rightarrow *$
 - instance pro `[]`, `BinTree`, `Maybe`
 - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)

Typová třída Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
 - instance pro `[]`, `BinTree`, `Maybe`
 - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou); `fmap` tento kontext nemění

Typová třída Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
 - instance pro `[]`, `BinTree`, `Maybe`
 - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou); `fmap` tento kontext nemění
- operátor `<$>` \equiv infixový `fmap`

```
  recip $ 2    ~> 0.5
```

```
  recip <$> Just 2 ~> Just 0.5
```

Pravidla pro třídu Functor

Instance třídy **Functor** musí splňovat dvě pravidla:

- `fmap id ≡ id`
- `fmap (f . g) ≡ fmap f . fmap g`

Pravidla pro třídu Functor

Instance třídy **Functor** musí splňovat dvě pravidla:

- `fmap id` \equiv `id`
- `fmap (f . g)` \equiv `fmap f . fmap g`

Pravidla musí platit!

- překladač se spoléhá na výše uvedená pravidla
- jejich platnost musí ověřit programátor (!)
- pro všechny knihovní instance platí

Instance třídy Functor I

- ```
instance Functor [] where
 fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
```

---

<sup>1</sup>Uvádět typy v instancích je dovoleno pouze s rozšířením InstanceSigs.  
Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

# Instance třídy Functor I

- ```
instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
fmap = map
```
- ```
instance Functor BinTree where
 fmap :: (a -> b) -> BinTree a -> BinTree b
```

---

<sup>1</sup>Uvádět typy v instancích je dovoleno pouze s rozšířením InstanceSigs.  
Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

# Instance třídy Functor I

- ```
instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
fmap = map
```
- ```
instance Functor BinTree where
 fmap :: (a -> b) -> BinTree a -> BinTree b
fmap = treeMap
```

---

<sup>1</sup>Uvádět typy v instancích je dovoleno pouze s rozšířením InstanceSigs.  
Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

# Instance třídy Functor II

- `instance Functor Maybe where`  
`fmap :: (a -> b) -> Maybe a -> Maybe b`

# Instance třídy Functor II

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

Instance třídy Functor II

- `instance Functor Maybe where`
 `fmap :: (a -> b) -> Maybe a -> Maybe b`
 `fmap f (Just x) = Just (f x)`
 `fmap f Nothing = Nothing`
- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (`Either e :: * -> *`)
 `instance Functor (Either e) where`
 `fmap :: (a -> b) -> Either e a -> Either e`
 `-> b`

Instance třídy Functor II

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

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (`Either e :: * -> *`)

```
instance Functor (Either e) where
    fmap :: (a -> b) -> Either e a -> Either e
        ↪ b
    fmap f (Right x) = Right (f x)
    fmap f (Left x)  = Left x
```

- Obdobně pro typový konstruktor dvojice:

```
instance Functor ((,) w) where
    fmap :: (a -> b) -> (w, a) -> (w, b)
```

Instance třídy Functor II

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

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (`Either e :: * -> *`)

```
instance Functor (Either e) where
    fmap :: (a -> b) -> Either e a -> Either e
        ↪ b
    fmap f (Right x) = Right (f x)
    fmap f (Left x)  = Left x
```

- Obdobně pro typový konstruktor dvojice:

```
instance Functor ((,) w) where
    fmap :: (a -> b) -> (w, a) -> (w, b)
    fmap f (x, y) = (x, f y)
```


Instance třídy Functor III

- ```
instance Functor IO where
 fmap :: (a -> b) -> IO a -> IO b
```

# Instance třídy Functor III

- ```
instance Functor IO where
    fmap :: (a -> b) -> IO a -> IO b
fmap f action = do
    result <- action
    pure (f result)
fmap' f action = action >>= (pure . f)
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor (\rightarrow) $:: * \rightarrow * \rightarrow *$

★ Instance třídy Functor IV

Funkce je binární typový konstruktor $(->)$ $:: * -> * -> *$

Částečná aplikace na jeden argument:

$(->) r :: * -> *$ (tedy „funkce z r “)

★ Instance třídy Functor IV

Funkce je binární typový konstruktor $(->)$ `:: * -> * -> *`

Částečná aplikace na jeden argument:

`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor $(->)$ `:: * -> * -> *`

Částečná aplikace na jeden argument:

$(->) r$ `:: * -> *` (tedy „funkce z r “)

```
instance Functor ((->) r) where
```

```
  fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor $(->)$ $:: * -> * -> *$

Částečná aplikace na jeden argument:

$(->) r :: * -> *$ (tedy „funkce z r “)

`instance Functor ((->) r) where`

`fmap :: (a -> b) -> (r -> a) -> (r -> b)`

`fmap f g = (\x -> f (g x)) -- === f . g`

★ Instance třídy Functor IV

Funkce je binární typový konstruktor $(->)$ `:: * -> * -> *`

Částečná aplikace na jeden argument:

$(->)$ `r` `:: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
```

```
  fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

```
  fmap f g = (\x -> f (g x))    -- === f . g
```

Intuice: hodnota závisí na kontextu (typu `r`), který teprve přijde.

```
greetings = (\polite -> if polite then "Good morning"
               else "Hello there")
```

```
ave = (++ " , Caesar.") <$> greetings
```