

Monády

IB016 Seminář z funkcionálního programování

Vypráví Adam Matoušek

Fakulta informatiky, Masarykova univerzita

Jaro 2021

Připomenutí: Maybe, Either, druhy

Maybe a: datový typ rozšířený o jednu hodnotu

- `data Maybe a = Nothing | Just a`
- často: `Nothing` je selhání

Připomenutí: Maybe, Either, druhy

Maybe a: datový typ rozšířený o jednu hodnotu

- `data Maybe a = Nothing | Just a`
- často: `Nothing` je selhání

Either a b: sjednocení dvou datových typů

- `data Either a b = Left a | Right b`
- často: specifikace chyby, nebo korektní výsledek

Připomenutí: Maybe, Either, druhy

Maybe a: datový typ rozšířený o jednu hodnotu

- `data Maybe a = Nothing | Just a`
- často: `Nothing` je selhání

Either a b: sjednocení dvou datových typů

- `data Either a b = Left a | Right b`
- často: specifikace chyby, nebo korektní výsledek

Druhy: „typování typů“

- `Maybe Int :: *`
- `Maybe :: * -> *`
- `Either :: * -> * -> *`

Připomenutí: Functor

Motivace: Zobecňování funkce `map`

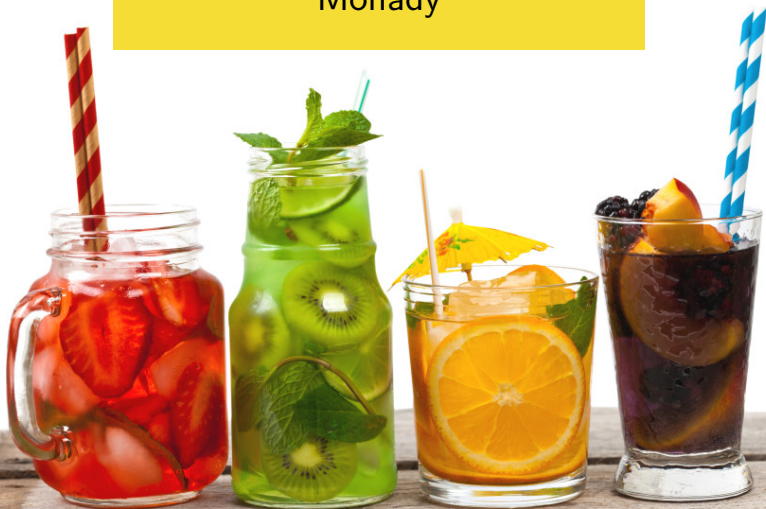
```
map :: (a -> b) -> [a] -> [b]
treeMap :: (a -> b) -> BinTree a -> BinTree b
. . . .
```

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 `[]`, `Tree`, `Maybe`, `IO`, `Either e`, `(->) r`, ...

Monády



Co je to monáda?

- zastaralý výraz pro prvoka
- termín z teorie kategorií
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

Co je to monáda?

- zastaralý výraz pro prvoka
- termín z teorie kategorií
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

Navrhovaná strategie pochopení monád:

- dívat se na různé funktory „výpočtovým pohledem“
- zkoumat u každého **zvlášť**, co znamená „řetěžit výpočet“
- abstrakce vyplyne časem sama

Výpočtový pohled na Maybe

Chceme umět sčítat (násobit, ...) hodnoty zabalené v **Just**:

```
mayOp :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
mayOp op (Just x) (Just y) = Just (x `op` y)
mayOp _ _ _ = Nothing
```

```
mayPlus = mayOp (+)
```

```
Just 42 `mayPlus` Just 24 ~>* Just 66
```

Výpočtový pohled na Maybe

Chceme umět sčítat (násobit, ...) hodnoty zabalené v **Just**:

```
mayOp :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
mayOp op (Just x) (Just y) = Just (x `op` y)
mayOp _ _ _ = Nothing
```

```
mayPlus = mayOp (+)
```

```
Just 42 `mayPlus` Just 24 ~>* Just 66
```

„Výpočtový pohled“:

- **Maybe** = možnost selhání výpočtu
- kombinujeme výpočet levého a výpočet pravého operandu
- výsledkem je výpočet součtu (součinu, ...)
- selže-li výpočet operandu, selhání se propaguje

Motivace I – Maybe

Výpočet, který může selhat:

- například `readInt :: String -> Maybe Int`

Výpočet, který může selhat:

- například `readInt :: String -> Maybe Int`

Skládání funkcí, které mohou selhat:

- skládání funkcí typu `Maybe a -> Maybe b` je jednoduché
- skládání funkcí typu `a -> Maybe b` už ne:

```
readAndDiv :: String -> String -> Maybe Int
```

```
readAndDiv s1 s2 =
```

```
  case readInt s1 of
```

```
    Nothing -> Nothing
```

```
    Just n1 -> case readInt s2 of
```

```
      Nothing -> Nothing
```

```
      Just 0 -> Nothing
```

```
      Just n2 -> Just (n1 `div` n2)
```

Zobecnění skládání

- kód pro skládání je nepřehledný
- jeho myšlenku lze ale zobecnit:
 - **Nothing** → přeskakujeme výpočet, vrátíme **Nothing**
 - **Just** x → pokračujeme ve výpočtu s hodnotou x

Zobecnění skládání

- kód pro skládání je nepřehledný
 - jeho myšlenku lze ale zobecnit:
 - **Nothing** → přeskakujeme výpočet, vrátíme **Nothing**
 - **Just** x → pokračujeme ve výpočtu s hodnotou x
- `bind :: Maybe a -> (a -> Maybe b) -> Maybe b`

Zobecnění skládání

- kód pro skládání je nepřehledný
- jeho myšlenku lze ale zobecnit:
 - **Nothing** → přeskakujeme výpočet, vrátíme **Nothing**
 - **Just** x → pokračujeme ve výpočtu s hodnotou x

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

`bind Nothing _ = Nothing`

Zobecnění skládání

- kód pro skládání je nepřehledný
- jeho myšlenku lze ale zobecnit:
 - **Nothing** → přeskakujeme výpočet, vrátíme **Nothing**
 - **Just** x → pokračujeme ve výpočtu s hodnotou x

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

```
bind Nothing _ = Nothing
```

```
bind (Just x) f = f x
```


Zobecnění skládání

- kód pro skládání je nepřehledný
- jeho myšlenku lze ale zobecnit:
 - `Nothing` → přeskakujeme výpočet, vrátíme `Nothing`
 - `Just x` → pokračujeme ve výpočtu s hodnotou `x`

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

```
bind Nothing _ = Nothing
```

```
bind (Just x) f = f x
```

Jednodušší řešení pro `readAndDiv`:

```
readAndDiv :: String -> String -> Maybe Int
```

```
readAndDiv s1 s2 =
```

```
  readInt s1 `bind` \n1 ->
```

```
  readInt s1 `bind` \n2 ->
```

```
  if n2 == 0 then Nothing
```

```
    else Just $ n1 `div` n2
```

Místo `bind` bychom mohli definovat přímo skládání.

- `compose` $:: (b \rightarrow \text{Maybe } c) \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow (a \rightarrow \text{Maybe } c)$

Motivace I – Maybe

Místo `bind` bychom mohli definovat přímo skládání.

- `compose` $:: (b \rightarrow \text{Maybe } c) \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow (a \rightarrow \text{Maybe } c)$

Lze ale jednoduše definovat pomocí `bind`:

- `compose f g = flip bind f . g`

Místo `bind` bychom mohli definovat přímo skládání.

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

Lze ale jednoduše definovat pomocí `bind`:

- `compose f g = flip bind f . g`

`bind` vs `compose`

- „zapojování“ funkcí pomocí `bind` může být přirozenější a přehlednější než skládání
- vyžaduje ale počáteční hodnotu typu `Maybe a`
- jak získat `Maybe a` z hodnoty typu `a`?

Místo `bind` bychom mohli definovat přímo skládání.

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

Lze ale jednoduše definovat pomocí `bind`:

- `compose f g = flip bind f . g`

`bind` vs `compose`

- „zapojování“ funkcí pomocí `bind` může být přirozenější a přehlednější než skládání
- vyžaduje ale počáteční hodnotu typu `Maybe` a
- jak získat `Maybe` a z hodnoty typu `a`?

```
unit :: a -> Maybe a
```

```
unit = Just
```

Chceme používat funkce, které mohou vracet seznam více hodnot stejného typu. Například:

- chceme všechny komplexní druhé/třetí odmocniny čísla
- máme různé varianty n-té otázky v odpovědníku
- získáváme obsah adresáře
- volíme směr v bludišti

Chceme používat funkce, které mohou vracet seznam více hodnot stejného typu. Například:

- chceme všechny komplexní druhé/třetí odmocniny čísla
- máme různé varianty n -té otázky v odpovědníku
- získáváme obsah adresáře
- volíme směr v bludišti

S každým mezivýsledkem má smysl zkusit pokračovat. Řetěžením takových funkcí tak můžeme například:

- zjistit všechny komplexní šesté odmocniny čísla
- generovat různé varianty celého odpovědníku
- rekurzivně získat všechny soubory v domovském adresáři
- backtrackingem najít východ z bludiště

Příklad:

- máme k dispozici funkce

```
-- 2. odmocniny
```

```
root2 :: Complex Float -> [Complex Float]
```

```
-- 3. odmocniny
```

```
root3 :: Complex Float -> [Complex Float]
```

- chceme funkci `root6` vracející všechny 6. odmocniny

Příklad:

- máme k dispozici funkce

```
-- 2. odmocniny
```

```
root2 :: Complex Float -> [Complex Float]
```

```
-- 3. odmocniny
```

```
root3 :: Complex Float -> [Complex Float]
```

- chceme funkci `root6` vracející všechny 6. odmocniny

```
root6 :: Complex Float -> [Complex Float]
```

```
root6 c = concat $ map root3 (root2 c)
```

Jak vypadají `bind` a `unit` pro nedeterministické výpočty?

Motivace II – Nedeterministické výpočty

Jak vypadají `bind` a `unit` pro nedeterministické výpočty?

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

```
bind list f = concat $ map f list
```

Jak vypadají `bind` a `unit` pro nedeterministické výpočty?

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

```
bind list f = concat $ map f list
```

```
unit :: a -> [a]
```

```
unit x = [x]
```

Pozorování: `[]` \approx **Nothing** (neexistuje žádný výsledek)

Motivace III – Náhodné funkce

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici stavové proměnné
- nutné předávat jako argument v celém výpočtu

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici stavové proměnné
- nutné předávat jako argument v celém výpočtu
- náhodné hodnoty můžeme reprezentovat jako funkce:
 - vstup je hodnota `seed`
 - výstup:
 - „náhodná“ hodnota závislá na `seed`
 - nová hodnota `seed'` pro další „náhodnou“ proměnnou
 - **Typ:**

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici stavové proměnné
- nutné předávat jako argument v celém výpočtu
- náhodné hodnoty můžeme reprezentovat jako funkce:
 - vstup je hodnota `seed`
 - výstup:
 - „náhodná“ hodnota závislá na `seed`
 - nová hodnota `seed'` pro další „náhodnou“ proměnnou
 - **Typ:** `Int -> (a, Int)`

Motivace III – Náhodné funkce

Pro přehlednost tento typ pojmenujme

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

Opět chceme řetězit výpočty.

Příklad:

- `rollDie :: Rand Int` na základě vstupní hodnoty vrátí číslo z intervalu $[1, 6]$
- jak vytvoříme výpočet, který háže 2 kostky po sobě a sečte výsledky?

Motivace III – Náhodné funkce

Pro přehlednost tento typ pojmenujme

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

Opět chceme řetězit výpočty.

Příklad:

- `rollDie :: Rand Int` na základě vstupní hodnoty vrátí číslo z intervalu $[1, 6]$
- jak vytvoříme výpočet, který háže 2 kostky po sobě a sečte výsledky?

```
roll2Dice :: Rand Int
```

```
roll2Dice = Rand $ \seed ->
```

```
  let
```

```
    (d1, seed') = runRand rollDie seed
```

```
    (d2, seed'') = runRand rollDie seed'
```

```
  in
```

```
    (d1 + d2, seed'')
```

Motivace III – Náhodné funkce

- opět je kompozice přirozená, ale nepřehledná
- zobecnění: levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodnou proměnnou na pravé straně.
- přesněji zachyceno pomocí `bind`

Motivace III – Náhodné funkce

- opět je kompozice přirozená, ale nepřehledná
- zobecnění: levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodnou proměnnou na pravé straně.
- přesněji zachyceno pomocí `bind`

```
bind :: Rand a -> (a -> Rand b) -> Rand b
bind x f = Rand $ \seed ->
  let (val, seed') = runRand x seed
  in runRand (f val) seed'
```

Motivace III – Náhodné funkce

- opět je kompozice přirozená, ale nepřehledná
- zobecnění: levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodnou proměnnou na pravé straně.
- přesněji zachyceno pomocí `bind`

```
bind :: Rand a -> (a -> Rand b) -> Rand b
```

```
bind x f = Rand $ \seed ->
```

```
  let (val, seed') = runRand x seed
```

```
  in runRand (f val) seed'
```

```
unit :: a -> Rand a
```

```
unit x = Rand $ \seed -> (x, seed)
```

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
```

```
roll2Dice = rollDie `bind` (\d1 ->  
    rollDie `bind` (\d2 -> unit (d1 + d2)))
```

★ Porovnání Rand a IO

Pro zájemce: přečtěte si **IO inside**. Ukazuje dobrou paralelu mezi **Rand** a **IO** a. K monádám s vnitřním stavem se ještě vrátíme v jiném cvičení.

Typové třídy Applicative, Monad

Typová třída Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- bind
  return :: a -> m a                -- unit
```


Typová třída Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- bind
  return :: a -> m a -- unit
```

```
(>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y
```

- umožňuje skládání výpočtů
 - předem definujeme, jak se toto skládání chová
 - `bind` navenek pracuje s výsledkem = „rozbalenou“ hodnotou
- definice třídy v Prelude, více v `Control.Monad`
- prozatím si nevímejme nadtrždy `Applicative...`

Instance třídy Monad I

```
instance Monad Maybe where
```

Instance třídy Monad I

```
instance Monad Maybe where
  return :: a -> Maybe a
  return = Just

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

- řetězení „parciálních“ funkcí (výpočet může selhat)
- >>=: výsledek se předává, pokud existuje

Instance třídy Monad I

```
instance Monad Maybe where
  return :: a -> Maybe a
  return = Just

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

- řetězení „parciálních“ funkcí (výpočet může selhat)
- >>=: výsledek se předává, pokud existuje
- jak vypadá instance pro **Either** e?

Instance třídy Monad II

```
instance Monad [] where
```

Instance třídy Monad II

```
instance Monad [] where
```

```
  return :: a -> [a]
```

```
  return x = [x]
```

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

```
  xs >>= f = concat (map f xs)
```

- řetězení „nedeterministických“ funkcí/výpočtů
(`a -> [b]` = jeden vstup, libovolný počet výsledků)
- `>>=`: výpočet probíhá nad každou hodnotou vstupu

Instance třídy Monad III

```
instance Monad ([1], _) where
```

Instance třídy Monad III

```
instance Monad ([], _) where
  return :: a -> ([], a)
  return x = ([], x)

  (>>=) :: ([], a) -> (a -> ([], b)) -> ([], b)
  (list, x) >>= f = let (list', y) = f x
                    in (list ++ list', y)
```

- řetězení funkcí/výpočtů s „logováním“
- >>=: udržuje se společný log celého výpočtu
- Spoiler alert! K této monádě se vrátíme v závěru kurzu.

Instance třídy Monad IV

```
instance Monad (r -> _) where -- funkce z r
```

Instance třídy Monad IV

```
instance Monad (r -> _) where -- funkce z r
  return :: a -> (r -> a)
  return x _ = x

  (>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
  (>>=) rx f ctx = let x = rx ctx
                    ry = f x
                    in ry ctx
```

- výpočty s read-only kontextem (např. konfigurace)
- >>=: výpočtu vstupní hodnoty i výpočtu výsledku předá týž kontext
- Spoiler alert! K této monádě se vrátíme v závěru kurzu.

Pravidla pro třídu Monad

- *levá identita*

`return x >>= f ≡ f x`

- *pravá identita*

`m >>= return ≡ m`

- *asociativita*

`(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)`

Užitečné funkce třídy Monad

```
(<=<) :: Monad m =>  
      (b -> m c) -> (a -> m b) -> (a -> m c)  
f <=< g = (\x -> g x >>= f)
```

- ekvivalent kompozice běžných funkcí (.)
- existuje i obrácená varianta >=>

```
join :: Monad m => m (m a) -> m a  
join mma = mma >>= id
```

- slučování vnořených kontextů
- ★ teoretici často místo `bindu` definují `join`

`do`-notation je jenom syntaktický cukr pro `>>=`

- tj. s každou monádou můžeme pracovat přes `do`¹
- ★ intensionální seznamy jsou cukrem kolem monád

¹Rozšíření `ApplicativeDo` umožňuje použít (omezené) `do` i s `Applicative`.

do-notation je jenom syntaktický cukr pro `>>=`

- tj. s každou monádou můžeme pracovat přes `do`¹

- ★ intensionální seznamy jsou cukrem kolem monád

```
maybePlus :: Num a => Maybe a -> Maybe a -> Maybe a
```

```
maybePlus x y = do
```

```
  justX <- x
```

```
  justY <- y
```

```
  return $ justX + justY
```

¹Rozšíření `ApplicativeDo` umožňuje použít (omezené) `do` i s `Applicative`.

Applicative

Motivace pro Applicative

Aplikativní funktory jsou na půli cesty mezi **Functor** a **Monad**.

- Umožňují z hodnoty vyrobit triviální výpočet:
 - `pure :: Applicative f => a -> f a`
 - doslova totéž co `return`, ale Historické Důvody™

Aplikativní funktory jsou na půli cesty mezi **Functor** a **Monad**.

- Umožňují z hodnoty vyrobit triviální výpočet:
 - `pure :: Applicative f => a -> f a`
 - doslova totéž co `return`, ale Historické Důvody™
- Umožňují kombinovat vícero výpočtů **čistými funkcemi**:
 - `liftA2 :: ... => (a -> b -> c) -> f a -> f b -> f c`
 - `liftA3` obdobně pro ternární
 - `<*> :: ... => f (a -> b) -> f a -> f b`
 - Nelze měnit *efekt* (`f`) na základě *výsledků* (`a`, `b`). Např.
`liftA2 op (Just 1) (Just 2)` neumí vrátit **Nothing**.

Existují aplikativní funktory, které nejsou monádami (**ZipList**).

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

- `ite = \i t e -> if i then t else e`

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

- `ite = \i t e -> if i then t else e`

```
ite True  io1 io2 ~>* 1, výstup One
```

```
ite False io1 io2 ~>* 2, výstup Two
```

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

- `ite = \i t e -> if i then t else e`

```
ite True  io1 io2 ~>* 1, výstup One
```

```
ite False io1 io2 ~>* 2, výstup Two
```

- `input = return True -- představuje uživ. vstup`
`liftA3 ite input io1 io2 ~>`

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

- `ite = \i t e -> if i then t else e`

```
ite True  io1 io2 ~>* 1, výstup One
```

```
ite False io1 io2 ~>* 2, výstup Two
```

- `input = return True -- představuje uživ. vstup`

```
liftA3 ite input io1 io2 ~> 1, výstup One\nTwo
```

- nejdříve se provedou všechny akce (vč. vedlejších efektů) a získají hodnoty (**True**, 1, 2). Až poté se jejich výsledky spojí funkcí `ite`

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:

```
io1 = putStrLn "One" >> return 1
```

```
io2 = putStrLn "Two" >> return 2
```

- `ite = \i t e -> if i then t else e`

```
ite True  io1 io2 ~>* 1, výstup One
```

```
ite False io1 io2 ~>* 2, výstup Two
```

- `input = return True -- představuje uživ. vstup`

```
liftA3 ite input io1 io2 ~> 1, výstup One\nTwo
```

- nejdříve se provedou všechny akce (vč. vedlejších efektů) a získají hodnoty (**True**, 1, 2). Až poté se jejich výsledky spojí funkcí `ite`
- potřebovali bychom vybalit z `input` čistý **Bool**...

★ Síla operátoru >>=

Podmíněné provádění akcí IO na základě uživatelského vstupu:

- mějme dvě akce typu **IO Int**:
`io1 = putStrLn "One" >> return 1`
`io2 = putStrLn "Two" >> return 2`
- `ite = \i t e -> if i then t else e`
`ite True io1 io2 ~>* 1`, výstup One
`ite False io1 io2 ~>* 2`, výstup Two
- `input = return True -- představuje uživ. vstup`
`liftA3 ite input io1 io2 ~> 1`, výstup One\nTwo
- nejdříve se provedou všechny akce (vč. vedlejších efektů) a získají hodnoty (**True**, 1, 2). Až poté se jejich výsledky spojí funkcí `ite`
- potřebovali bychom vybalit z `input` čistý **Bool**... a předat ho funkci vracející akci IO

Typová třída `Applicative`

```
class Functor f => Applicative f where
  pure    :: a -> f a
  (<*>)   :: f (a -> b) -> f a -> f b
  liftA2  :: (a -> b -> c) -> f a -> f b -> f c
```

- samotná třída v `Prelude`, více v `Control.Applicative`
- předepsaných funkcí je více, ale dají se odvodit
- stačí jedno z `(<*>)` a `liftA2`
- lze napsat instanci `Monad` a použít `liftA2 = liftM2`
- naopak `return` se automaticky zadefinuje jako `pure`

Applicative vs. Monad

Z Historických Důvodů™ je spousta „aplikativních“ věcí v **Monad**:

- `pure` \equiv `return`
- `(<*>)` \equiv `ap`
- `(*>)` \equiv `(>>)`
- `liftA` \equiv `liftM` (\equiv `fmap`)
- `liftA2` \equiv `liftM2`
- `liftA3` \equiv `liftM3`

Applicative má navíc `(<*>)`, **Monad** zase až `liftM5`

Pravidla pro třídu Applicative

- *identita*

`(pure id) <*> x ≡ x`

- *kompozice*

`(pure (.)) <*> f <*> g <*> x ≡ f <*> (g <*> x)`

- *homomorfismus*

`(pure f) <*> (pure x) ≡ pure (f x)`

- *výměna*

`u <*> (pure y) ≡ (pure ($) <*> u)`

```
class Monad m => MonadFail m where  
    fail :: String -> m a
```

- akce `fail` má přerušit probíhající výpočet
- historicky (ještě v GHC 8.6) součástí třídy `Monad`
- automaticky se používá při pattern-matchingu v `do`-bloku
- instance pro `[]`, `Maybe` a `IO`

Funktory a monády – shrnutí

Functor:

- `fmap :: (a -> b) -> f a -> f b`:
- aplikace funkce na výsledek výpočtu

Applicative:

- `pure :: a -> f a`
- triviální výpočet se zadaným výsledkem
- `liftA2 :: (a -> b -> c) -> f a -> f b -> f c`
- kombinace výsledků nezávislých výpočtů

Monad:

- `(>>=) :: m a -> (a -> m b) -> m b`
- výpočet může záviset na výsledku předchozího výpočtu
- `join :: m (m a) -> m a`
- lze spustit výpočet, který je výsledkem výpočtu