Druhy, funktory, monády

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Rest z minula

Typový konstruktor `Either`

```haskell
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

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

Připomenutí: Typové třídy

Typové třídy

Typová třída = kolekce jmen funkcí + typů

```
class MakesSound a where
    doSound :: a -> String
```

Typové třídy

Typová třída = kolekce jmen funkcí + typů

```
class MakesSound a where
    doSound :: a -> String
```

Instance typové třídy = implementace požadovaných funkcí pro daný typ

```
data Animal = Dog | Cat
data MoneyPrinter = MP { amount :: Int }

instance MakesSound Animal where
    doSound Dog = "haf"
    doSound Cat = "mnau"

instance MakesSound MoneyPrinter where
    doSound mp = "b" ++ replicate (amount mp) 'r'
```

Typové třídy

Typová třída = kolekce jmen funkcí + typů

```
class MakesSound a where
    doSound :: a -> String
```

Instance typové třídy = implementace požadovaných funkcí pro daný typ

```
data Animal = Dog | Cat
data MoneyPrinter = MP { amount :: Int }

instance MakesSound Animal where
    doSound Dog = "haf"
    doSound Cat = "mnau"

instance MakesSound MoneyPrinter where
    doSound mp = "b" ++ replicate (amount mp) 'r'
```

Typové třídy jde použít k omezení typových parametrů:

```
appendSound :: MakesSound a => a -> [String] -> [String]
appendSound thing sounds = doSound thing : sounds
```

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

```haskell
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`)

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 `<$>` ≡ 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` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla pro třídu Functor

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

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

[1] 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
  ```

[1] 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
  ```

[1] Uvádět typy v instancích je dovoleno pouze s rozšířením `InstanceSigs`. Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

Vlastní instance třídy Functor II

Mějme vlastní verzi datového typu **Maybe** a:

```
data MyMaybe a = MyNothing | MyJust a deriving (Show)
```

Napište pro **MyMaybe** instanci typové třídy **Functor**.

Instance třídy Functor II

- **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

- **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

- **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

```haskell
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 `(->) :: * -> * -> *`

★ 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
```

Monády

Monády

Co je to monáda?

- ~~zastaralý výraz pro prvoka~~
- ~~základní substance, ze které se skládá vesmír~~[1]
- ~~termín z teorie kategorií~~
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

[1] https://en.wikipedia.org/wiki/Monad_(philosophy)

Monády

Co je to monáda?

- ~~zastaralý výraz pro prvoka~~
- ~~základní substance, ze které se skládá vesmír~~[1]
- ~~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 „řetězit výpočet“
- abstrakce vyplyne časem sama

[1] https://en.wikipedia.org/wiki/Monad_(philosophy)

Výpočty a jejich výsledky

Výpočet $\approx$ hodnota $\approx$ výsledek vyhodnocení $\Rightarrow$ výsledek výpočtu

```
x :: Int
x = 19, ale i
x = succ $ succ (2 * 4) * 2
```

Výpočty a jejich výsledky

Výpočet $\approx$ hodnota $\approx$ výsledek vyhodnocení $\Rightarrow$ výsledek výpočtu

```
x :: Int
x = 19, ale i
x = succ $ succ (2 * 4) * 2

y :: Maybe String
y = Just "adamat" nebo y = Nothing, ale i
y = lookup 445763 (getStudents db)
```

(Výsledkem výpočtu je `"adamat"`, nebo žádný neexistuje.)

Maybe jako výpočet

`Maybe` a je výpočet s možností selhání (parciální funkce):

- $\rightsquigarrow^*$ **`Just`** `42` — výsledkem je 42
- $\rightsquigarrow^*$ **`Nothing`** — výsledek není, výpočet selhal

Maybe jako výpočet

Maybe a je výpočet s možností selhání (parciální funkce):

- $\rightsquigarrow^*$ **Just** 42 — výsledkem je 42
- $\rightsquigarrow^*$ **Nothing** — výsledek není, výpočet selhal

Máme parciální výpočty

- `x, y :: Maybe Int`, které vrátí číslo nebo selžou,
- `f :: Int -> Maybe Int`, která z čísla vypočítá číslo nebo selže,
- `g :: String -> Maybe Int`, která z řetězce vypočítá číslo nebo selže.

Maybe jako výpočet

Maybe a je výpočet s možností selhání (parciální funkce):

- $\rightsquigarrow^*$ `Just 42` — výsledkem je 42
- $\rightsquigarrow^*$ **Nothing** — výsledek není, výpočet selhal

Máme parciální výpočty

- `x, y :: Maybe Int`, které vrátí číslo nebo selžou,
- `f :: Int -> Maybe Int`, která z čísla vypočítá číslo nebo selže,
- `g :: String -> Maybe Int`, která z řetězce vypočítá číslo nebo selže.

Jak vyrobit výpočty, které by intuitivně odpovídaly

- `x + y :: Maybe Int`
- `f x :: Maybe Int`
- `f . g :: String -> Maybe Int`

Ani jeden z těchto výrazů není typově korektní. ☹

Kombinace výpočtů v Maybe

```
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

Kombinace výpočtů pomocí (čisté[2]) funkce:

- provedeme výpočet „operandů“ (podvýpočty),
- pokud některý selže, i kombinace selže (**`Nothing`**),
- jinak se výsledkem (v **`Just`**) kombinace podvýsledků.

Kombinací dostaneme zase výpočet (tj. něco v **`Maybe`**).

[2] ve smyslu „není sama monadickým výpočtem“[3]
[3] konkrétně zde „nebalí výsledek do Maybe“

Kombinace výpočtů v Maybe

```
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

Kombinace výpočtů pomocí (čisté[2]) funkce:

- provedeme výpočet „operandů“ (podvýpočty),
- pokud některý selže, i kombinace selže (**Nothing**),
- jinak se výsledkem (v **Just**) kombinace podvýsledků.

Kombinací dostaneme zase výpočet (tj. něco v **Maybe**).

Příklad:

```
liftM2 (+) (Just 10) (Just 2) ⇝* Just 12
liftM2 (+) (Nothing) (Just 2) ⇝* Nothing
```

[2] ve smyslu „není sama monadickým výpočtem“[3]

[3] konkrétně zde „nebalí výsledek do Maybe“

Řetězení výpočtů v Maybe

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

Řetězení výpočtů:

- provedeme výpočet mezivýsledku,
- pokud se povedl (`Just x`), pokračujeme dál s hodnotou `x`
- pokud selhal (`Nothing`), nepokračujeme ($\rightsquigarrow^*$ `Nothing`)

Řetězení výpočtů v Maybe

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

Řetězení výpočtů:

- provedeme výpočet mezivýsledku,
- pokud se povedl (**Just** x), pokračujeme dál s hodnotou x
- pokud selhal (**Nothing**), nepokračujeme ($\rightsquigarrow^*$ **Nothing**)

Příklad:

```
tryRead "3" >>= \x -> safeDiv 10 (x - 1) ⇝* Just 5
tryRead "a" >>= \x -> safeDiv 10 (x - 1) ⇝* Nothing
tryRead "1" >>= \x -> safeDiv 10 (x - 1) ⇝* Nothing
```

Triviální výpočet

- Na totální funkci můžeme nahlížet jako na parciální.
- S neselhávajícím výpočtem můžeme pracovat, jako by selhání umožňoval.
- Pro hodnotu umíme vyrobit výpočet, který ji prostě vrací.

Triviální výpočet

- Na totální funkci můžeme nahlížet jako na parciální.
- S neselhávajícím výpočtem můžeme pracovat, jako by selhání umožňoval.
- Pro hodnotu umíme vyrobit výpočet, který ji prostě vrací.

- ```
  pure :: a -> Maybe a
  pure x = Just x
  ```
  ```

Maybe je monáda

1. Umíme řetězit výpočty:
   `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b`
2. Umíme vyrobit triviální výpočet hodnoty:
   `pure :: a -> Maybe a`

= podstata monády[4]

[4] obě funkce ještě musí splňovat nějaká pravidla, uvidíme později

Seznam = nedeterministické výpočty

`[a]` je výpočet s možností více výsledků (nedeterministická funkce):

- $\rightsquigarrow^*$ `[42]` — výsledkem je 42
- $\rightsquigarrow^*$ `[42, 12]` — výsledkem je 42 nebo 12
- $\rightsquigarrow^*$ `[]` — výsledek není žádný

Seznam = nedeterministické výpočty

`[a]` je výpočet s možností více výsledků (nedeterministická funkce):

- $\rightsquigarrow^*$ `[42]` — výsledkem je 42
- $\rightsquigarrow^*$ `[42, 12]` — výsledkem je 42 nebo 12
- $\rightsquigarrow^*$ `[]` — výsledek není žádný

Máme nedeterministické výpočty

- `x, y :: [Int]`, které vrátí několik možných čísel,
- `f :: Int -> [Int]`, která z čísla vypočítá několik možných čísel
- `g :: String -> [Int]`, která z řetězce vypočítá několik možných čísel

Seznam = nedeterministické výpočty

`[a]` je výpočet s možností více výsledků (nedeterministická funkce):

- $\rightsquigarrow^*$ `[42]` — výsledkem je 42
- $\rightsquigarrow^*$ `[42, 12]` — výsledkem je 42 nebo 12
- $\rightsquigarrow^*$ `[]` — výsledek není žádný

Máme nedeterministické výpočty

- `x, y :: [Int]`, které vrátí několik možných čísel,
- `f :: Int -> [Int]`, která z čísla vypočítá několik možných čísel
- `g :: String -> [Int]`, která z řetězce vypočítá několik možných čísel

Pomocí monády `[a]` jde vyrobit výpočty, které odpovídají

- `x + y :: [Int]`
- `f x :: [Int]`
- `f . g :: String -> [Int]`

Seznam = nedeterministické výpočty

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

Seznam = nedeterministické výpočty

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. Řetězením 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

Seznam = nedeterministické výpočty

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

Seznam = nedeterministické výpočty

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)
```

Seznam = nedeterministické výpočty

Jak vypadá řetězení a `pure` pro nedeterministické výpočty?

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

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

Seznam = nedeterministické výpočty

Jak vypadá řetězení a `pure` pro nedeterministické výpočty?

```
(>>=) :: [a] -> (a -> [b]) -> [b]
list >>= f = concat $ map f list

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

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

Výpočet s náhodou

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

Výpočet s náhodou

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na seed → potřebujeme seed propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální 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:**

Výpočet s náhodou

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na seed → potřebujeme seed propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální 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)`

Výpočet s náhodou

Pro přehlednost tento typ pojmenujme

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

Opět chceme řetězit výpočty.
Příklad:

- `rollDie :: Rand Int` vrátí náhodné číslo z intervalu $[1, 6]$
- `roll2Dice :: Rand Int`

Výpočet s náhodou

Pro přehlednost tento typ pojmenujme

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

Opět chceme řetězit výpočty.
Příklad:

- `rollDie :: Rand Int` vrátí náhodné číslo z intervalu $[1, 6]$
- ```
  roll2Dice :: Rand Int
  roll2Dice = Rand $ \seed ->
      let
          (d1, seed')  = runRand rollDie seed
          (d2, seed'') = runRand rollDie seed'
      in
          (d1 + d2, seed'')
  ```
  ```

Výpočet s náhodou

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

- Princip řetězení náhodných výpočtů zobecníme.
- Levá strana bere seed → vytváří nový seed' → použije se jako vstup pro náhodné číslo na pravé straně.

```haskell
(>>=) :: Rand a -> (a -> Rand b) -> Rand b
```

```haskell
pure :: a -> Rand a
```

Výpočet s náhodou

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

- Princip řetězení náhodných výpočtů zobecníme.
- Levá strana bere seed → vytváří nový seed' → použije se jako vstup pro náhodné číslo na pravé straně.

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

pure :: a -> Rand a
pure x = Rand $ \seed -> (x, seed)
```

Výpočet s náhodou

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
roll2Dice =
    rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
    pure (d1 + d2)))
```

Výpočet s náhodou

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
roll2Dice =
    rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
    pure (d1 + d2)))
```

Nebo pomocí **do**-notace:

```
roll2Dice :: Rand Int
roll2Dice = do
    d1 <- rollDie
    d2 <- rollDie
    return (d1 + d2)
```

Výpočet s náhodou

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
roll2Dice =
    rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
    pure (d1 + d2)))
```

Nebo pomocí **do**-notace:

```
roll2Dice :: Rand Int
roll2Dice = do
    d1 <- rollDie
    d2 <- rollDie
    return (d1 + d2)
```

Nebo pomocí `liftM2`:

```
roll2Dice :: Rand Int
roll2Dice = liftM2 (+) rollDie rollDie
```

Rand jako monáda s vnitřním stavem

Obecněji: monády s vnitřním stavem.

- **`Rand`** — stav je sémě generátoru
- **`Turtle`** — poloha a směr želvy
- **`Parser`** — stav obsahuje nepřečtenou část textu
- **`State`** s — vlastní libovolný stav typu s
- ★ **`IO`** — stav je kouzelný[5]

(Některé uvidíme později během semestru.)

[5] Pro zájemce: https://wiki.haskell.org/IO_inside

Typová třída Monad

Typová třída Monad

```haskell
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b  -- čti „bind“
    return :: a -> m a                 -- = pure
```

Typová třída Monad

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b  -- čti „bind“
    return :: a -> m a                 -- = pure

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

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

Instance třídy Monad I

```haskell
instance Monad Maybe where
    return :: a -> Maybe a

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

Instance třídy Monad I

```haskell
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
```

- „Parciální“ funkce (výpočet může selhat).
- `>>=`: výsledek se předává, pokud existuje.

Instance třídy Monad II

```haskell
instance Monad [] where
    return :: a -> [a]

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

Instance třídy Monad II

```
instance Monad [] where
    return :: a -> [a]
    return x = [x]

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

- „Nedeterministické“ funkce/výpočty
  (`a -> [b]` = jeden vstup, libovolný počet výsledků).
- `>>=`: výpočet probíhá nad každou hodnotou vstupu

Instance třídy Monad III

```haskell
instance Monad ([l], _) where – dvojice se seznamem
    return :: a -> ([l], a)

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

Instance třídy Monad III

```haskell
instance Monad ([l], _) where – dvojice se seznamem
    return :: a -> ([l], a)
    return x = ([], x)

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

- Výpočty s „logováním“.
- >>=: udržuje se společný log celého výpočtu.
- Obecněji: monáda písaře. Uvidíme později v semestru.

Instance třídy Monad IV

```haskell
instance Monad (r -> _) where -- fukce z r
    return :: a -> (r -> a)

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

Instance třídy Monad IV

```
instance Monad (r -> _) where -- fukce 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).
- >>=: všem výpočtům předá týž kontext
- Monáda čtenáře. Uvidíme ještě později v semestru.

Pravidla pro třídu Monad

Korektní instance monády musí splňovat:

- *levá identita*
  `return x >>= f ≡ f x`
- *pravá identita*
  `mx >>= return ≡ mx`
- *asociativita*
  `(mx >>= f) >>= g ≡ mx >>= (\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 mmx = mmx >>= id
```

- slučování vnořených kontextů
- ★ teoretici často místo `bind`u definují `join`

Samostatné programování

Příklad: monáda Maybe

Mějme následující typ reprezentující aritmetické výrazy:

```
data Expr = Const Int |
            Add Expr Expr | Sub Expr Expr |
            Mul Expr Expr | Div Expr Expr
            deriving (Show)
```

Naprogramujte funkci `eval :: Expr -> Maybe Int`, která vyhodnotí zadaný výraz a *ošetří dělení nulou*.

Poté přidejte datovému typu **`Expr`** ještě `if-then-else` hodnotový konstruktor **`IfZero Expr Expr`** a odpovídajícím způsobem upravte funkci `eval` .

Příklad: skládání více výpočtů

Na prvním semináři jste viděli funkci `sequence :: [IO a] -> IO [a]`, která vyrábí ze seznamu výpočtů jeden výpočet, který vrací seznam výsledků všech těchto výpočtů.

Knihovní funkce `sequence` je ve skutečnosti obecnější:

```
sequence :: Monad m => [m a] -> m [a]
```

Co tato funkce dělá pro monády `Maybe` a `List`?

Až to zjistíte, naprogramujte vlastní funkci `mySequence :: Monad m => [m a] -> m [a]`.

Příklad: vlastní monáda

Mějme vlastní definici typu `Either a b` a odpovídající instance `Functor` a `Applicative`[6].

```
import Control.Applicative

data MyEither a b = MyLeft a | MyRight b deriving Show

instance Functor (MyEither a) where
  fmap f (MyLeft l) = MyLeft l
  fmap f (MyRight r) = MyRight (f r)

instance Applicative (MyEither a) where
  pure x = MyRight x
  liftA2 f (MyRight r1) (MyRight r2) = MyRight (f r1 r2)
  liftA2 f (MyLeft l1) _ = MyLeft l1
  liftA2 f _ (MyLeft l2) = MyLeft l2
```

Naprogramujte pro `MyEither` instanci třídy `Monad`. Jakým výpočtům tato monáda odpovídá?

[6] Zatím neřešte, co je zač třída `Applicative`.

Příklad: ekvivalence definic monády

Typová třída `Monad m` vyžaduje buď definice funkcí

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

nebo definice funkcí

```
return :: m a
join :: m (m a) -> m a
```

Ukažte, že tyto definice jsou ekvivalentní. Tj.,

- pomocí `return` a `(>>=)` jde vyrobit `join`, a
- pomocí `return` a `join` jde vyrobit `(>>=)`.

Nápověda: Možná se vám bude hodit, že třída `Monad` je podtřídou `Functor`.