Cvičení 7

7.1 Typové třídy

Příklad 7.1.1 Vysvětlete, co je to typová třída a jak může programátorovi použití typové třídy pomoci (ušetřit práci) při tvorbě a definici vlastních typů.

Příklad 7.1.2 Uvažte datový typ představující semafor zadefinovaný níže.

```
data TrafficLight = Red | Orange | Green
```

Umožněte zobrazování hodnot tohoto typu, jejich vzájemné porovnávání a řazení (zelená < oranžová < červená). Řečeno jinak, napište instanci `TrafficLight` pro typové třídy `Show`, `Eq` a `Ord`.

Příklad 7.1.3 Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární datový konstruktor `PairD` (viz definice níže).

```
data PairT a b = PairD a b
```

Vytvořte instanci `PairT` pro typové třídy `Show`, `Eq` a `Ord`. Ať jsou si dvě dvojice rovny právě tehdy, pokud jsou si rovny po složkách. Uspořádání použijte lexikografické. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1,2)` vypište třeba `"pair of 1 and 2"`).

Příklad 7.1.4 Vytvořte speciální verzi podmíněného výrazu s názvem `iff`, který bude mít polymorfní podmínku (tedy nevyžaduje striktně typ `Bool`). Vytvořte si pomocnou typovou třídu `Boolable`, která bude sdružovat všechny typy, které se dají interpretovat jako `Bool`. Typová třída bude zaručovat implementaci funkce `getBool :: Boolable a => a -> Bool`. Vytvořte instance pro několik základních typů (`Bool`, `Int`, `[a]`, ...).

Příklad 7.1.5 Jaký je rozdíl mezi těmito dvěma definicemi? Předpokládejte datový typ `Nat` zavedený jako `data Nat = Zero | Succ Nat`.

- ```
  instance Ord Nat where
    (<=) Zero (Succ _) = True
    ...
  ```
- ```
  (<=) Zero (Succ _) = True
  ...
  ```

7.2 Akumulační funkce na seznamech

Příklad 7.2.1 Definujte následující funkce rekurzivně:

a) `product'` – součin prvků seznamu
b) `length'` – počet prvků seznamu
c) `map'` – funkci `map`

Co mají tyto definice společné? Jak by vypadalo jejich zobecnění?

Příklad 7.2.2 Určete, co dělají akumulační funkce s uvedenými argumenty. Najděte hodnoty, na které je lze tyto výrazy aplikovat, a ověřte pomocí interpretu.

a) `foldr (+) 0`
b) `foldr1 (\x s -> x + 10 * s)`
c) `foldl1 (\s x -> 10 * s + x)`

Příklad 7.2.3 Definujte funkci `subtractlist`, která odečte druhý a všechny další prvky *neprázdného* seznamu od jeho prvního prvku, tj. `subtractlist [`x_1`, ..., `x_n`] = `x_1 ` - `x_2 ` - ... - `x_n.

Příklad 7.2.4 Předpokládejme, že funkce `compose` skládá všechny funkce ze seznamu funkcí, tj. `compose [`f_1`, ..., `f_n`] = `f_1 ` . ... . `f_n.

a) Definujte funkci `compose` s využitím akumulátorových funkcí.
b) Jaký je typ funkce `compose`? (Odvoďte bez použití interpretru a poté ověřte.)

Příklad 7.2.5 Uvažme funkci: `foldr (.) id`

a) Jaký je význam uvažované funkce?
b) Jaký je její typ?
c) Uveďte příklad částečné aplikace této funkce na jeden argument.
d) Uveďte příklad úplné aplikace této funkce na kompletní seznam argumentů.

Příklad 7.2.6 Jaký je význam a typ funkce `foldr (:)`?

Příklad 7.2.7 Jaký je význam a typ funkce `foldl (flip (:)) []`?

Příklad 7.2.8 Vaší úlohou je implementovat funkci dle specifikace v zadání za použití standardních funkcí `foldr`, `foldl`, `foldr1`, `foldl1`. Požadovaný typ funkce je vždy uveden. Pokud není řečeno jinak, řešení by nemělo obsahovat formální parametry – má tedy být v následujícím tvaru:

```
functionName = foldr (function) (term)
```

Jestliže je možné příklad řešit více než jednou z nabízených akumulačních funkcí, vyberte tu, která je nejefektivnější. K většině zadání je dostupný i ukázkový výsledek na jednom seznamu sloužící jako ilustrace.

Každé řešení zapisujte i s typem funkce, v některých příkladech by bez něj nemuselo jít zkompilovat (důvod je poněkud složitější).

a) Funkce `sumFold` vrátí součet čísel v zadaném seznamu.

```
sumFold :: Num a => [a] -> a
sumFold [1,2,4,5,7,6,2] ⇝* 27
```

b) Funkce `productFold` vrátí součin čísel v zadaném seznamu.

```
productFold :: Num a => [a] -> a
productFold [1,2,4,0,7,6,2] ⇝* 0
```

c) Funkce `orFold` vrátí `True`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `True`, jinak vrátí `False`.

```
orFold :: [Bool] -> Bool
orFold [False, True, False] ⇝* True
```

d) Funkce `andFold` vrátí `False`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `False`, jinak vrátí `True`.

```
andFold :: [Bool] -> Bool
andFold [False, True, False] ⇝* False
```

e) Funkce `lengthFold` vrátí délku zadaného seznamu.

```
lengthFold :: [a] -> Int
lengthFold [1,2,3,5,8,0] ⇝* 6
```

f) Funkce `minimumFold` vrátí minimální prvek ze zadaného neprázdného seznamu.

```
minimumFold :: Ord a => [a] -> a
minimumFold ['f','e','r','t'] ⇝* 'e'
```

g) Funkce `maximumFold` vrátí maximální prvek ze zadaného neprázdného seznamu.

```
maximumFold :: Ord a => [a] -> a
maximumFold ['f','e','r','t'] ⇝* 't'
```

h) Funkce `maxminFold` vrátí minimální a maximální prvek ze zadaného seznamu ve formě uspořádané dvojice. Použijte i formální argument seznam, budete ho potřebovat při definici. Funkce by měla projít zadany seznam pouze jednou!

```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold [1,5,7,2,6,8,2] ⇝* (1,8)
```

i) Funkce `composeFold` vezme seznam funkcí a hodnotu, a vrátí hodnotu, která vznikne postupným aplikováním funkcí v seznamu na danou hodnotu (poslední funkce se aplikuje jako první, první jako poslední).

```
composeFold :: [a -> a] -> a -> a
composeFold [(*8),(+2),(flip mod 5)] 27 ⇝* 32
```

j) Funkce `idFold` vrátí zadaný seznam beze změny.

```
idFold :: [a] -> [a]
idFold [5,3,2,1,4,1] ⇝* [5,3,2,1,4,1]
```

k) Funkce `concatFold` vrátí zřetězení prvků zadaného seznamu seznamů.

```
concatFold :: [[a]] -> [a]
concatFold [[1,2],[3],[4,5]] ⇝* [1,2,3,4,5]
```

l) Funkce `listifyFold` nahradí každý prvek jednoprvkovým seznamem s původním prvkem.

```
listifyFold :: [a] -> [[a]]
listifyFold [1,3,4,5] ⇝* [[1],[3],[4],[5]]
```

m) Funkce `nullFold` vrátí `True`, pokud je zadaný seznam prázdný, jinak vrátí `False`.

```
nullFold :: [a] -> Bool
nullFold [1,2,5,6] ⇝* False
```

n) Funkce `headFold` vrátí první prvek zadaného neprázdného seznamu.

```
headFold :: [a] -> a
headFold [2,1,5,4,8] ⇝* 2
```

o) Funkce `lastFold` vrátí poslední prvek zadaného neprázdného seznamu.

```
lastFold :: [a] -> a
lastFold [2,1,5,7,8] ⇝* 8
```

p) Funkce `reverseFold` vrátí zadaný seznam s prvky v obráceném pořadí.

```
reverseFold :: [a] -> [a]
reverseFold "asdfghj" ⇝* "jhgfdsa"
```

q) Funkce `suffixFold` vrátí seznam všech přípon zadaného seznamu (jako první bude samotný seznam, poslední bude prázdný seznam).

```
suffixFold :: [a] -> [[a]]
suffixFold "abcd" ⇝* ["abcd","bcd","cd","d",""]
```

r) Funkce `mapFold` vezme funkci a seznam, a vrátí seznam, který vznikne aplikací zadané funkce na každý prvek zadaného seznamu. Pro funkci použijte formální argument.

```
mapFold :: (a -> b) -> [a] -> [b]
mapFold (+5) [1,2,3] ⇝* [6,7,8]
```

s) Funkce `filterFold` vezme predikát a seznam a vrátí seznam, který vznikne ze zadaného seznamu vyloučením všech prvků, na kterých predikát vrátí `False`. Pro predikát použijte formální argument.

```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold odd [1,2,4,8,6,2,5,1,3] ⇝* [1,5,1,3]
```

t) Funkce `oddEvenFold` vrátí v uspořádané dvojici seznamy prvků z lichých a sudých pozic původního seznamu.

```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold [1,2,7,5,4] ⇝* ([1,7,4], [2,5])
```

u) Funkce `takeWhileFold` vezme predikát a seznam, a vrátí nejdelší prefix seznamu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold even [2,4,1,2,4,5,8,6,8] ⇝* [2,4]
```

v) Funkce `dropWhileFold` vezme predikát a seznam, a vrátí zadaný seznam bez nejdelšího prefixu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold odd [1,2,5,9,1,7,4,6] ⇝* [2,5,9,1,7,4,6]
```

Příklad 7.2.9 Definujte funkci `foldl` pomocí funkce `foldr`.

Příklad 7.2.10 Je možné definovat funkci `f` tak, aby se `foldr f [] s` vyhodnotilo na seznam obsahující jenom prvky ze sudých míst v seznamu `s`?

Je možné definovat takovou funkci `f` pro použití ve výrazu `snd $ foldl f (True, []) s`?

Příklad 7.2.11 Proč je implementace funkce `or` (logická disjunkce všech hodnot v seznamu) pomocí funkce `foldr` lepší než pomocí `foldl`?

Příklad 7.2.12 Mějme funkci `foldr2` definovanou následovně:

```
foldr2 :: (a -> a -> b -> b) -> (a -> b) -> b -> [a] -> b
foldr2 f2 f1 f0 [] = f0
foldr2 f2 f1 f0 [x] = f1 x
foldr2 f2 f1 f0 (x:y:s) = f2 x y (foldr2 f2 f1 f0 s)
```

Zkuste definovat funkci `foldr` pomocí `foldr2` a funkci `foldr2` pomocí `foldr`, nebo zdůvodněte, proč to není možné.

Řešení

Řešení 7.1.1 Typová třída umožňuje deklarovat určitou vlastnost datového typu na základě definice několika funkcí. Například to, že je na datovém typu definována rovnost, uspořádání a nebo že jeho hodnoty je možné převést do řetězcové reprezentace. Výhodou používání typových tříd je možnost používat stejné intuitivní operátory nebo funkce jako pro jiné typy a taky možnost získat na základě definovaných základních funkcí i další funkce, které lze vyjádřit pomocí základních. U některých jednoduchých případů datových typů a jednodušších typových tříd je možné i automaticky odvodit definice základních funkcí (!), k čemuž se používá klauzule `deriving` následována n-ticí typových tříd, pro které mají být příslušné funkce automaticky odvozeny.

Řešení 7.1.2

```
instance Show TrafficLight where
  show Red      = "Red light!"
  show Orange   = "Orange light!"
  show Green    = "Green light!"
instance Eq TrafficLight where
  Red      == Red      = True
  Orange   == Orange   = True
  Green    == Green    = True
  _        == _        = False
instance Ord TrafficLight where
  _        <= Red      = True
  Orange   <= Orange   = True
  Green    <= _        = True
  _        <= _        = False
```

Instanci typové třídy `Ord` lze s použitím `(==)` definovat i na tři řádky:

```
instance Ord TrafficLight where
  _        <= Red      = True
  Green    <= _        = True
  x        <= y        = x == y
```

Řešení 7.1.3

```
instance (Eq a, Eq b) => Eq (PairT a b) where
  (PairD x y) == (PairD v w) = (x == v) && (y == w)
instance (Show a, Show b) => Show (PairT a b) where
  show (PairD x y) = "pair of " ++ show x ++ " and " ++ show y
instance (Ord a, Ord b) => Ord (PairT a b) where
  (PairD x y) <= (PairD v w) = x < v || (x == v && y <= w)
```

Řešení 7.1.4

```
iff :: Boolable b => b -> a -> a -> a
iff b t e = if getBool b then t else e

class Boolable a where
  getBool :: a -> Bool

instance Boolable Bool where
  getBool x = x

instance Boolable Int where
  getBool 0 = False
  getBool _ = True

instance Boolable [a] where
  getBool [] = False
  getBool _ = True
```

Řešení 7.1.5 V prvním případu jde o zavedení operátoru `(<=)` na datovém typu `Nat`. Tento operátor bude tedy možné používat na typech jako doposud plus na typu `Nat`.

Ve druhém případě jde o redefinici operátoru `(<=)`. Tento operátor bude tedy možné používat jenom na typu `Nat`. Původní definice bude překryta touto a nebude dostupná přímo. (Ale bude dostupná při uvedení kvalifikovaného názvu operátoru: `(Prelude.<=) 3 4` nebo `3 Prelude.<= 4`)

Řešení 7.2.1

a)
```
product' :: Num a => [a] -> a
product' []    = 1
product' (x:s) = x * product' s
```

b)
```
length' :: [a] -> Int
length' []    = 0
length' (_:s) = 1 + length' s
```

c)
```
map' :: (a -> b) -> [a] -> [b]
map' _ []    = []
map' f (x:s) = f x : map' f s
```

Vždy jde o definici, která vrací určitou hodnotu na prázdném seznamu. V případě neprázdného seznamu je zas vrácen výsledek nějaké funkce, která bere jako argument první prvek a rekurzivní volání stejné funkce (i s případnými argumenty jako u `map'`) na zbytku seznamu.

Jejich funkcionalitu lze tedy abstrahovat na funkci, která dostane jako jeden argument hodnotu vracenou na prázdném seznamu a jako druhý argument funkci, která se aplikuje na první prvek neprázdného seznamu a na výsledek volání požadované funkce na zbytku seznamu. Tedy:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []    = z
foldr f z (x:s) = f x (foldr f z s)
```

Řešení 7.2.2

a) Funkce vloží mezi každý člen seznamu operátor `+` a na konec seznamu zařadí `0`. Tedy lze ji aplikovat na číselný seznam a výsledkem je součet hodnot jeho prvků, tj. odpovídá funkci `sum`.

b) Nejsnáze se význam tohoto výrazu objasní na příkladě. Na základě typu λ-abstrakce vidíme, že funkce pracuje na číslech.

```
foldr1 (\x s -> x + 10 * s) [1,2,3] ⇝* 1 + 10 * (2 + 10 * 3)
    ⇝* 321
```

c) Tento případ je podobný jako u výrazu s `foldr1`. Opět se podívejme na příklad vyhodnocení:

```
foldl1 (\s x -> 10 * s + x) [1,2,3] ⇝* 10 * (10 * 1 + 2) + 3
    ⇝* 123
```

Tedy funkce převede seznam čísel reprezentující dekadický rozklad čísla na jedno číslo.

Řešení 7.2.3 Jasným kandidátem na řešení je použití některé z akumulačních funkcí. Celkem máme na výběr ze čtyř: `foldr, foldl, foldr1, foldl1`. Připomeňme si, jak která z nich funguje:

```
foldr  (@) w [1,2,3,4] = 1 @ (2 @ (3 @ (4 @ w)))
foldr1 (@)   [1,2,3,4] = 1 @ (2 @ (3 @ 4))
foldl  (@) w [1,2,3,4] = (((w @ 1) @ 2) @ 3) @ 4
foldl1 (@)   [1,2,3,4] = ((1 @ 2) @ 3) @ 4
```

Na základě těchto příkladů vidíme, že jediným vhodným kandidátem na přirozenou definici funkce `subtractlist` je `foldl1`. Tedy ve výsledku dostaneme:

```
subtractlist :: [Integer] -> Integer
subtractlist = foldl1 (-)
```

Řešení 7.2.4

a) `compose = foldr (.) id`
V tomto případě by fungovalo i `compose = foldl (.) id`, protože operace skládání je asociativní. Ale přirozenou asociativitou pro ni je asociativita zprava, proto jsme použili `foldr`.

b) Viz příklad 7.2.5.

Řešení 7.2.5

a) Funkce `foldr`, jak je nám známo, pracuje na seznamech a nahrazuje `(:)` za funkci, v tomto případě za `(.)`, a `[]` za `id`. Intuitivně musí jít o seznam funkcí, které budeme postupně skládat. Ve výsledku tedy vytvoříme složení funkcí v pořadí, v jakém jsou uvedeny v seznamu.

b) Při určování typu lze postupovat následovně algoritmicky:
- Máme daný výraz `foldr (.) id`
- Zjistíme si typy všech funkcí:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (a -> b) -> (c -> a) -> c -> b
id    :: a -> a
```

- Přejmenujeme typové proměnné, aby měl typ každé funkce vlastní typové proměnné:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (d -> e) -> (f -> d) -> f -> e
id    :: c -> c
```

- Určíme typové rovnosti na základě aplikací:

```
(d -> e) -> (f -> d) -> f -> e = a -> b -> b
c -> c = b
```

- Rozepíšeme typové rovnosti do jednodušších:

```
d -> e = a
f -> d = b
f -> e = b
c -> c = b
```

- Vyjádříme si všechny proměnné pomocí co nejmenšího počtu proměnných:

```
d = e = f = c
b = c -> c
a = c -> c
```

- Zjistíme, jaký typ vlastně hledáme. Je to typový výraz odpovídající typu `foldr` s odstraněnými dvěma typovými argumenty, tedy ve výsledku `[a] -> b`. Dosadíme do něj vyjádření proměnných získaných v předchozím kroku a tím dostaneme výsledný typ:

```
foldr (.) id :: [a] -> b = [c -> c] -> c -> c
```

c) `foldr (.) id [(+4), (*10), (42^)]`
d) `foldr (.) id [(+4), (*10), (42^)] 1`

Řešení 7.2.6 Nejprve si výraz upravíme na `\z s -> foldr (:) z s`. Vzpomeňme si, že `foldr` nahrazuje výskyty `(:)` a `[]`. V tomto případě výskyty `(:)` neměníme. Nahradíme jenom výskyt prázdného seznamu na konci. Intuitivně, prázdný seznam můžeme nahradit libovolným seznamem s prvky stejného typu, jako má vstupní seznam.

Když si pak představíme, co bude takováto funkce vracet, bude to seznam obsahující nejprve prvky seznamového argumentu funkce `foldr` a pak na místo `[]` dosadíme seznam `z`. Tedy výraz `foldr (:) z s` vrací stejný výsledek jako výraz `s ++ z`. Je tedy ekvivalentní výrazu `flip (++)`.

Po odvození typu výrazu dostaneme `[a] -> [a] -> [a]`.

Řešení 7.2.7 Víme, že funkce `foldl` „skládá“ seznam zleva. Podívejme se, jak dochází k vyhodnocování této funkce na krátkém seznamu:

```
   foldl (flip (:)) [] [1,2,3]
⇝ flip (:) (flip (:) (flip (:) [] 1) 2) 3
⇝ 3 : (flip (:) (flip (:) [] 1) 2)
⇝ 3 : (2 : (flip (:) [] 1))
```

```
⇝ 3 : (2 : (1 : []))
≡  [3,2,1]
```

Vidíme tedy, že tato funkce funguje jako funkce `reverse`.

Typ funkce je `[a] -> [a]`.

Řešení 7.2.8 Pro některé podúlohy je uvedeno více ekvivalentních řešení – tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

a)
```
sumFold :: Num a => [a] -> a
sumFold = foldr (+) 0
sumFold' = foldr (\e t -> e + t) 0
```

b)
```
productFold :: Num a => [a] -> a
productFold = foldr (*) 1
productFold' = foldr (\e t -> e * t) 1
```

c)
```
orFold :: [Bool] -> Bool
orFold = foldr (||) False
orFold' = foldr (\e t -> e || t) False
```

d)
```
andFold :: [Bool] -> Bool
andFold = foldr (&&) True
andFold' = foldr (\e t -> e && t) True
```

e)
```
lengthFold :: [a] -> Int
lengthFold = foldr (\_ t -> t + 1) 0
```

f)
```
minimumFold :: Ord a => [a] -> a
minimumFold = foldr1 min
minimumFold' = foldr1 (\e t -> if e < t then e else t)
minimumFold'' list = foldr min (head list) list
```

g)
```
maximumFold :: Ord a => [a] -> a
maximumFold = foldr1 max
maximumFold' = foldr1 (\e t -> if e < t then t else e)
maximumFold'' list = foldr max (head list) list
```

h)
```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold list = foldr (\e (tMin, tMax) -> (min e tMin, max e tMax) )
    (head list, head list) list
```

i)
```
composeFold :: [(a -> a)] -> a -> a
composeFold = foldr (.) id
composeFold' = flip (foldr id)
```

j)
```
idFold :: [a] -> [a]
idFold = foldr (:) []
idFold' = foldr (\e t -> e : t) []
```

k)
```
concatFold :: [[a]] -> [a]
concatFold = foldr (++) []
```

l)
```
listifyFold :: [a] -> [[a]]
listifyFold = foldr (\x s -> [x]:s) []
listifyFold' = foldr ((:) . (:[])) []
```

m)
```
nullFold :: [a] -> Bool
nullFold = foldr (\_ _ -> False) True
```

n)
```
headFold :: [a] -> a
headFold = foldr1 const
headFold' = foldr1 (\e t -> e)
```

o)
```
lastFold :: [a] -> a
lastFold = foldr1 (flip const)
lastFold' = foldr1 (\e t -> t)
```

p)
```
reverseFold :: [a] ->  [a]
reverseFold = foldl (flip (:)) []
reverseFold' = foldl (\t e -> e : t) []
reverseFold'' = foldr (\e t -> t ++ [e]) []
```

Poslední řešení má výrazně vyšší složitost!

q)
```
suffixFold :: [a] -> [[a]]
suffixFold = foldr (\e (x:xs) -> (e:x) : x : xs) [[]]
```

r)
```
mapFold :: (a -> b) -> [a] -> [b]
mapFold f = foldr (\e t -> f e : t) []
```

s)
```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr (\e t -> if p e then e:t else t) []
```

t)
```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold = foldr (\x (l, r) -> (x:r, l)) ([], [])
```

u)
```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold p = foldr (\e t -> if p e then e:t else []) []
```

v)
```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold p = foldl (\t e -> if null t && p e then [] else t ++ [e])
   []
dropWhileFold' p list = foldl (\t e -> if null (t []) && p e then id
   else t.(e:) ) id list []
```

Druhé uvedené řešení má lepší složitost.

Řešení 7.2.9

```
foldl f z s = foldr (flip f) z (reverse s)
```

Řešení 7.2.10 Ne, není to možné. Funkce `f` by musela dokázat rozlišit, kdy je volána na prvku ze sudého místa a kdy z lichého. K dispozici má však pouze n-tý prvek a zbytek seznamu od $(n+1)$-tého prvku. Pokud bychom předpokládali, že tato funkce existuje, musela by fungovat korektně i na dvojici seznamů `[1,2,3]` a `[0,1,2,3]`. Avšak v jistém okamžiku bude volána

tak, že dostane jako argument hodnotu `1` a výsledek výrazu `foldr f [] [2,3]`. Pak ale nelze rozeznat, o který případ se jedná, přičemž v jednom má vrátit `[1,3]` a v druhém `[0,2]`.

Ve druhém případě to možné je:

```
f = \(b, s) x -> (not b, if b then s ++ [x] else s)
```

Teď již postupujeme zleva (`foldl`) a v hodnotě typu `Bool` si ukládáme, jestli je aktuální pozice sudá.

Řešení 7.2.11 Funkce `foldr` přestane vyhodnocovat výraz po prvním nalezeném `True` (díky vlastnosti funkce `(||)`, která se nazývá „short-circuiting“), avšak `foldl` ho vždy projde celý. Tedy v případě nekonečného seznamu `foldr` skončí po prvním nalezeném `True`, ale `foldl` neskončí nikdy.

Řešení 7.2.12

```
foldr f z s = foldr2 (\x y s -> f x (f y s)) (\x -> f x z) z s
```

Opačná definice (`foldr2` pomocí `foldr`) není možná. Pomocí `foldr2` je možné vybrat každý druhý prvek seznamu (`foldr2 (\x y s -> x:s) (:[]) []`), což však pomocí `foldr` není možné (viz úloha 7.2.10).