Cvičení 8

8.1 Akumulační funkce nad vlastními datovými typy

Příklad 8.1.1 Mějme datový typ `Nat` reprezentující přirozená čísla:

```
data Nat = Zero | Succ Nat
```

Definujte funkci `natfold`, která je tzv. katamorfismem na typu `Nat` (tj. je funkcí, která nahrazuje všechny datové konstruktory datového typu, stejně jako je akumulační funkce `foldr` seznamovým katamorfismem).

```
natfold :: (a -> a) -> a -> Nat -> a
```

Příklady zamýšleného použití funkce `nfold`:

1. Funkce `natfold (Succ . Succ) Zero :: Nat -> Nat` „zdvojnásobuje“ hodnotu typu `Nat`.
2. Funkce `natfold (1+) 0 :: Nat -> Int` převádí hodnotu typu `Nat` do celých čísel typu `Int`.

Příklad 8.1.2 Vaší úlohou je implementovat funkce dle zadaných specifikací s použitím funkce `treeFold` zadané níže. Požadovaný typ funkce je vždy uveden. Jestli úloha neříká jinak, řešení by mělo být bez formálních parametrů, tedy v následujícím tvaru:

```
functionName = treeFold (function) (term)
```

Na pořadí zpracovávání jednotlivých uzlů nezáleží, ideálně však zpracujte vždy nejdříve levý podstrom, pak samotný vrchol a na závěr pravý podstrom (v některých podúlohách se bude výsledek lišit dle pořadí zpracovávání).

Úlohy řešte pro binární stromy typu `BinTree a` používané na cvičení. Jejich definice spolu s definicí „foldovací“ funkce `treeFold` jsou pro úplnost uvedené níže. Naleznete je také v souboru `08_treeFold.hs` v ISu a v příloze sbírky.

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving Show
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold f e Empty = e
treeFold f e (Node v l r) = f v (treeFold f e l) (treeFold f e r)
```

Ke většině úloh je dostupný i ukázkový výsledek na předem zvoleném stromě (slouží jako ilustrace, co zadání vlastně požaduje). Kvůli přehlednosti jsou ukázkové stromy pojmenované a jejich úplný tvar najdete až za poslední podúlohou.

a) Funkce `treeSum` vrátí součet čísel ve všech uzlech zadaného stromu.

```
treeSum :: Num a => BinTree a -> a
treeSum tree01 ⇝* 16
```

b) Funkce `treeProduct` vrátí součin čísel ve všech uzlech zadaného stromu.

```
treeProduct :: Num a => BinTree a -> a
treeProduct tree01 ⇝* 120
```

c) Funkce `treeOr` vrátí `True`, jestli se v zadaném stromě nachází alespoň jedenkrát hodnota `True`, jinak vrátí `False`.

```
treeOr :: BinTree Bool -> Bool
treeOr tree05 ⇝* True
```

d) Funkce `treeSize` vrátí počet uzlů v zadaném stromě.

```
treeSize :: BinTree a -> Int
treeSize tree01 ⇝* 6
treeSize tree06 ⇝* 5
```

e) Funkce `treeHeight` vrátí výšku zadaného stromu (poznámka: prázdný strom má výšku 0, jednouzlový strom má výšku 1).

```
treeHeight :: BinTree a -> Int
treeHeight tree03 ⇝* 2
treeHeight tree01 ⇝* 3
```

f) Funkce `treeList` vrátí seznam hodnot ze všech uzlů. Nejdříve uveďte hodnoty z levého podstromu, pak hodnotu v uzlu a následně hodnoty z pravého podstromu (tzv. *inorder* procházení stromu).

```
treeList :: BinTree a -> [a]
treeList tree01 ⇝* [5,3,2,1,4,1]
treeList tree02 ⇝* ["A","B","C","D","E"]
```

g) Funkce `treeConcat` vrátí zřetězení hodnot ze všech uzlů.

```
treeConcat :: BinTree [a] -> [a]
treeConcat tree02 ⇝* "ABCDE"
```

h) Funkce `treeMax` vrátí maximální hodnotu ze všech hodnot v uzlech. Hodnoty musí být z typové třídy `Ord` a `Bounded` (poznámka: zkuste funkce `minBound` a `maxBound`). Upozornění: Stromy, které budete používat na vyhodnocování mějte explicitně otypovány, jinak můžete narazit na problém při kompilaci (důvod je poněkud složitější).

```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax tree01 ⇝* 5
treeMax tree03 ⇝* (3,3)
```

i) Funkce `treeFlip` vrátí zadaný strom, avšak každá jeho pravá větev bude vyměněna s příslušnou levou větví.

```
treeFlip :: BinTree a -> BinTree a
treeFlip tree01 ⇝* Node 2 (Node 4 (Node 1 Empty Empty)
                                  (Node 1 Empty Empty))
                          (Node 3 Empty (Node 5 Empty Empty))
treeConcat (treeFlip tree02) ⇝* "EDCBA"
```

j) Funkce `treeId` vrátí zadaný strom v nezměněné podobě (Pozor! Stále vyžadujeme použití funkce `treeFold`!).

```
treeId :: BinTree a -> BinTree a
treeId tree05 ⇝* tree05
```

k) Funkce `rightMostBranch` vrátí seznam hodnot nejpravější větve zadaného stromu (v nejpravější větvi nikdy „nezatáčíme doleva“).

```
rightMostBranch :: BinTree a -> [a]
rightMostBranch tree01 ⇝* [2,4,1]
rightMostBranch tree02 ⇝* ["C","E"]
```

l) Funkce `treeRoot` vrátí kořenový prvek zadaného stromu. Jestli je strom prázdný, program havaruje (poznámka: můžete použít hodnotu `undefined`).

```
treeRoot :: BinTree a -> a
treeRoot tree01 ⇝* 2
```

m) Funkce `treeNull` zjistí, jestli je zadaný strom prázdný (podobá se funkci `null` pro seznamy).

```
treeNull :: BinTree a -> Bool
treeNull tree01 ⇝* False
treeNull tree04 ⇝* True
```

n) Funkce `leavesCount` vrátí počet listů v zadaném stromě (list je každý uzel, který nemá potomky).

```
leavesCount :: BinTree a -> Int
leavesCount tree01 ⇝* 3
leavesCount tree04 ⇝* 0
```

o) Funkce `leavesList` vrátí seznam hodnot z listů zadaného stromu. Preferované pořadí listů v seznamu je zleva doprava.

```
leavesList :: BinTree a -> [a]
leavesList tree01 ⇝* [5,1,1]
leavesList tree02 ⇝* ["B","D"]
```

p) Funkce `treeMap` aplikuje zadanou funkci na hodnotu v každém uzlu zadaného stromu (poznámka: funkce pracuje podobně jako `map` na seznamech). Výsledná funkce může mít jeden formální parametr.

```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeSum (treeMap (+1) tree01) ⇝* 22
```

q) Funkce `treeAny` zjistí, jestli alespoň jedna hodnota v zadaném stromě splňuje zadaný predikát (tedy se na něm vyhodnotí na `True`). Výsledná funkce může mít jeden formální parametr.

```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny (==10) tree01 ⇝* False
treeAny even tree01 ⇝* True
treeAny null tree02 ⇝* False
```

r) Funkce `treePair` zjistí, jestli je v každém uzlu stromu první složka uspořádané dvojice rovná druhé složce této dvojice.

```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair tree03 ⇝* False
```

s) Funkce `subtreeSums` vloží do každého uzlu zadaného stromu součet všech uzlů podstromu určeného tímto uzlem.

```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums tree01 ⇝* Node 16 (Node 8 (Node 5 Empty Empty) Empty)
```

```
                                    (Node 6 (Node 1 Empty Empty)
                                            (Node 1 Empty Empty))
```

t) Funkce `findPredicates` vezme 2 argumenty: základní hodnotu a strom, který má v každém uzlu uspořádanou dvojici tvořenou „identifikačním“ číslem a predikátem. Úlohou je vrátit seznam čísel odpovídajících predikátům, které se na dané základní hodnotě vyhodnotí na `True`. Výsledná funkce může mít jeden formální parametr.

```
findPredicates :: a -> BinTree (Int, a -> Bool) -> [Int]
findPredicates 3 tree06 ⇝* [1,3,4]
findPredicates 6 tree06 ⇝* [0,4]
```

Ukázkové stromy:

```
tree01 :: BinTree Int
tree01 = Node 2 (Node 3 (Node 5 Empty Empty) Empty)
                (Node 4 (Node 1 Empty Empty) (Node 1 Empty Empty))

tree02 :: BinTree String
tree02 = Node "C" (Node "A" Empty (Node "B" Empty Empty))
                  (Node "E" (Node "D" Empty Empty) Empty)

tree03 :: BinTree (Int,Int)
tree03 = Node (3,3) (Node (2,1) Empty Empty)
                    (Node (1,1) Empty Empty)

tree04 :: BinTree a
tree04 = Empty

tree05 :: BinTree Bool
tree05 = Node False (Node False Empty (Node True Empty Empty))
                    (Node False Empty Empty)

tree06 :: BinTree (Int, Int -> Bool)
tree06 = Node (0,even) (Node (1,odd) (Node (2,(== 1)) Empty Empty) Empty)
                       (Node (3,(< 5)) Empty (Node (4,((== 0) . mod 12))
                        Empty Empty))
```

Příklad 8.1.3 Uvažte datový typ `NTree a` definovaný níže a nad ním definovanou funkci `ntreeFold`:

```
data NTree a = NNode a [NTree a]
             deriving (Show, Read)
ntreeFold :: (a -> [b] -> b) -> NTree a -> b
ntreeFold f (NNode v ts) = f v (map (ntreeFold f) ts)
```

S pomocí této funkce (tedy ve tvaru `foo = ntreeFold ...`) definujte následující funkce:

a) `ntreeSize :: NTree a -> Integer`, která spočítá počet uzlů ve stromě
b) `ntreeMax :: Ord a => NTree a -> a`, která spočítá maximum z ohodnocení daného stromu

c) `ntreeDepth :: NTree a -> Integer`, která spočítá hloubku stromu (počet uzlů na nejdelší cestě z kořene do listu)

d) `ntreeMap :: (a -> b) -> NTree a -> NTree b`, která na hodnocení každého uzlu ve svém druhém parametru aplikuje funkci, která je jejím prvním parametrem:

```
ntreeMap (+1) (NNode 0 [NNode 1 [], NNode 41 []])
    ⇝* NNode 1 [NNode 2 [], NNode 42 []]
```

8.2 Opakování

Příklad 8.2.1 Otypujte následující výrazy:

a)
```
head [id, not]
```

b)
```
\f -> f 42
```

c)
```
\t x -> x + x > t x
```

d)
```
\xs -> filter (> 2) xs
```

e)
```
\f -> map f [1,2,3]
```

f)
```
foo f True  = map f [1,2,3]
foo f False = filter f [1,2,3]
```

g)
```
\(p,q) z  ->  q (tail z) : p (head z)
```

Příklad 8.2.2 Uvažte následující datový typ:

```
data Foo a = Bar [a]
           | Baz a
```

Určete typy následujících výrazů:

a)
```
getlist (Bar xs) = xs
getlist (Baz x ) = x : []
```

b)
```
\foo -> foldr (+) 0 (getlist foo)
```

Příklad 8.2.3 Definujte funkci `minmax :: Ord a => [a] -> (a, a)`, která pro daný neprázdný seznam *v jednom průchodu* spočítá minimum i maximum.

Funkci definujte jednou rekurzivně a jednou pomocí `foldr` nebo `foldl` (ne `foldr1`, `foldl1`).

Dále implementujte funkci `minmaxBounded :: (Ord a, Bounded a) => [a] -> (a, a)`, která funguje i na prázdných seznamech. Využijte konstant `minBound :: Bounded a => a`, `maxBound :: Bounded a => a`.

Najděte datový typ, pro který `minmaxBounded` nefunguje.

Poznámka: Může se stát, že interpretr bude zmatený z požadavku `Bounded a` a nebude schopen sám vyhodnotit výrazy jako `minmaxBounded [1,2,3]`. V takovém případě explicitně určete typ prvků seznamu, například: `minmaxBounded [1,2,3::Int]`.

Příklad 8.2.4 Převeďte následující funkce do pointwise tvaru a přepište je s pomocí intensionálních seznamů a bez použití funkcí `map, filter, curry, uncurry, zip, zipWith`.

a) `map . uncurry`
b) `\f xs -> zipWith (curry f) xs xs`
c) `map (* 2) . filter odd . map (* 3) . map (`div` 2)`
d) `map (\f -> f 5) . map (+)`

Příklad 8.2.5 Otypujte následující `IO` výrazy a převeďte je do `do`-notace: Uvažujte při tom následující typy `(>>=)`, `(>>)`, `return`:

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

a) `readFile "/etc/passwd" >> putStrLn "bla"`
b) `\f -> putStrLn "bla" >>= f`
c) `getLine >>= \x -> return (read x)`
d)
```
foo :: Integer
foo = getLine >>= \x -> read x
```

Příklad 8.2.6 Které z následujících výrazů jsou korektní?

a) `max a b + max (a - b) (b + a) (b * a)`
b) `False < True || True`
c) `5.1 ^ 3.2 ^ 2`
d) `2 ^ if even m then 1 else m`
e) `m mod 2 + 11 / m`
f) `(/) 3 2 + 2`
g) `[(4,5),(7,8),(1,2,3)]`
h) `(\s -> s, s)`
i) `[ x | x <- [1..10], odd x, let m = 5 * x - 1]`
j) `('a':"bcd", "a":"bcd", "a":"bcd":[])`
k) `[] : map f x`
l) `map f x : []`
m) `fst (map, 3) fst []`
n) `[[1],[2]..[10]]`
o) `\t -> if t == (x:_) then x else 0`
p) `\x s -> fst (x:s) : repeat x`
q) `[ [] | _ <- [] ]`
r) `[ m * n | m <- [1..] | n <- iterate (^2) 1 ]`
s) `zip [10,20..] [1,4,9,16,..]`
t) `getLine >>= putStrLn`
u) `getLine >> putStrLn`
v) `\t -> getLine >> putStrLn t`
w) `(>>) (putStrLn "OK") . putStrLn`
x) `x >>= (f >>= g) >>= h`

Příklad 8.2.7 Které z následujících typů jsou korektní?

a) `[Int, Int]`
b) `(Int)`

c) `[()] -> ([])`
d) `(Show x) => [x]`
e) `a -> b -> c -> d -> e`
f) `(A -> b) -> [A] -> b`
g) `String -> []`
h) `IO (String -> IO ())`
i) `IO (Maybe Int)`
j) `IO a -> IO a`
k) `[string] -> int -> bool -> string`
l) `Int a => b`
m) `a -> (Int b => b)`
n) `(Integral a, Num a) => a`
o) `Num a -> Num a`
p) `Num -> c -> c`

Příklad 8.2.8 Vyhodnoťte následující výrazy (zjednodušte do co nejjednoduššího tvaru).

a) `init [1] ++ [2]`
b) `head []`
c) `concat []`
d) `map f []`
e) `map (map (0:)) [[]]`
f) `map (++[]) [[[], []], [], [[]]]`
g) `[ 0 | False ]`
h) `[ [] | _ <- [[]] ]`
i) `[ [[]] | _ <- [] ]`
j) `(\x -> 3 * x + (\x -> x + x ^ 2) (2 * x - 1)) 5`
k) `(\f x -> f id (max 5) x) (.) 3`
l) `[] ++ map f (x ++ [])`

Příklad 8.2.9 Které z následujících úprav jsou korektní?

a) `(+1) (*2)` ⇝ `(+1) . (*2)`
b) `f . (.g)` ⇝ `\x -> f (.g x)`
c) `getLine >>= \x -> putStrLn (reverse x) >> putStrLn "done"` ⇝
`(getLine) >>= (\x -> putStrLn (reverse x)) >> (putStrLn "done")`
d) `\x y -> (\z -> f z) + 3` ⇝ `\x y z -> f z + 3`
e) `and (zipWith (==) s1 s2) && False` ⇝* `False`

Příklad 8.2.10 Otypujte následující výrazy:

a) `[]`
b) `[()]`
c) `tail [True]`
d) `(id.)`
e) `flip id`
f) `id (id id (id id)) ((id id) id)`
g) `(.) (.)`
h) `map flip`

i) `(.) . ((.).)`
j) `\g -> g (:) []`
k) `\s -> [t | (h:t) <- s, h]`
l) `\x y -> (x y, y x)`
m) `\[] -> []`
n) `(>>getLine)`
o)
```
do  x <- getLine
    let y = reverse x
    putStrLn ("reverted: " ++ y)
```

Příklad 8.2.11 Definujte funkci `nth :: Int -> [a] -> [a]`, která vybere každý n-tý prvek ze seznamu (seznam začíná nultým prvkem, můžete předpokládat, že $n \geq 1$). Zkuste úlohu vyřešit několika způsoby. Příklad: `nth 3 [1..10]` $\rightsquigarrow^*$ `[1,4,7,10]`.

Příklad 8.2.12 Definujte funkci `modpwr`, která funguje stejně jako funkce `\n k m -> mod (n^k) m`, ale implementujte ji efektivně (tak, aby v průběhu výpočtu nevycházela čísla, která jsou větší než `n`^{3}). Můžete předpokládat, že `k` $\geq$ `0, m` $\geq$ `1`. Funkce by měla mít logaritmickou časovou složitost vzhledem na velikost `k`.

Pomůcka: Použijte techniku *exponentiation by squaring* a dělejte zbytky už z mezivýsledků.

Příklad 8.2.13 Co dělají následující funkce?

a) `f1 = flip id 0`
b) `f2 = flip (:) []`
c) `f3 = zipWith const`
d) `f4 p = if p then ('/':) else id`
e) `f5 = foldr id 0`
f) `f6 = foldr (const not) True`

Příklad 8.2.14 Které z následujících vztahů jsou *obecně* platné (tj. platí pro každou volbu argumentů)? Uvažte také typ výrazů. Neplatné vztahy zkuste opravit.

a) `reverse (s ++ [x])` $\equiv$ `x : reverse s`
b) `map f . filter p` $\equiv$ `filter p . map f`
c) `flip . flip` $\equiv$ `id`
d) `foldr f (foldr f z s) t` $\equiv$ `foldr f z (s ++ t)`
e) `sum (zipWith (+) m n)` $\equiv$ `sum m + sum n`
f) `head s : tail s` $\equiv$ `s`
g) `map f (iterate f x)` $\equiv$ `iterate f (f x)`
h) `foldr (1+) 0` $\equiv$ `length`

Příklad 8.2.15 V následujících výrazech doplňte vhodný podvýraz za `...` tak, aby ekvivalence platily *obecně* (tedy pro libovolnou volbu proměnných).

a) `foldr ... z s` $\equiv$ `foldr f z (map g s)`
b) `zipWith ... x y` $\equiv$ `map f (zipWith g (map h1 x) (map h2 y))`
c) `foldl ... z s` $\equiv$ `foldl f z (filter p s)`
d) `foldr ... [] (s1, s2)` $\equiv$ `zip s1 s2`

e) `foldr ...` ≡ `const :: a -> [b] -> a`

Příklad 8.2.16 Jakou časovou složitost má vyhodnocení následujících výrazů? Uvažujte normální redukční strategii. Určete jenom asymptotickou složitost (konstantní, lineární, kvadratická, ..., exponenciální, výpočet nekončí). Předpokládejte, že použité proměnné jsou již plně vyhodnoceny a jejich hodnotu lze získat v jednom kroku.

a) `head s` (vzhledem k délce `s`)
b) `sum s` (vzhledem k délce `s`, pro jednoduchost předpokládejte, že operace sčítaní má konstantní složitost)
c) `take m [1..]` (vzhledem k hodnotě `m`)
d) `take m [1..10^6]` (vzhledem k hodnotě `m`)
e) `take n [1..m]` (vzhledem k hodnotám `m`, `n`)
f) `m ++ n` (vzhledem k délkám seznamů `m`, `n`)
g) `repeat x` (vzhledem k celočíselné hodnotě `x`)
h) `head (head s : tail s)` (vzhledem k délce seznamu `s`)
i) `fst (n, sum [1.0..n] / n)` (vzhledem k hodnotě `n`)

8.3 Složitější příklady

Příklad 8.3.1 Vaší úlohou je napsat program na řešení Hanojských věží. Tento matematický hlavolam vypadá následovně: Máme tři kolíky (věže), očíslujme si je 1, 2, 3. Na začátku máme na kolíku 1 všech N disků ($N \geq 1$) s různými poloměry postavených tak, že největší disk je naspodu a nejmenší navrchu. Vaší úlohou je přemístit disky na kolík 2 tak, aby byly stejně seřazené. Během celého procesu však musíme dodržet 3 pravidla:

- Disky přemísťujeme po tazích po jednom.
- Tah spočívá v tom, že vezmeme kotouč, který je na vrcholu některé věže a položíme ho na vrchol jiné věže.
- Je zakázáno položit větší kotouč na menší.

Napište funkci, která dostane počet disků, číslo kolíku, na kterém se na začátku disky nachází, a číslo kolíku, kam je chceme přesunout. Funkce vrátí seznam uspořádaných dvojic `(a,b)` – tahů, kde `a` je číslo kolíku, ze kterého jsme přesunuli vrchní disk na kolík `b`. Tedy například `hanoi 3 1 2` vrátí `[(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)]`.

Příklad 8.3.2 Dokažte, že funkce `($) . ($) . ... . ($)` představuje pro libovolný konečný počet `($)` vždy tu stejnou funkci. Zkuste intuitivně argumentovat, proč tomu tak je.

Příklad 8.3.3 Z funkcí nacházejících se v modulu `Prelude` vytvořte bez použití podmíněné konstrukce (`if`) funkci `if'` splňující `if' b x y = if b then x else y`

Příklad 8.3.4 Zkuste přepsat následující výraz pomocí intensionálních seznamů a funkcí pracujících se seznamy:

```
if cond1 then val1 else if cond2 then val2 else ... else val_default
```

Příklad 8.3.5 Co nejpřesněji popište, co dělají následující funkce.

a)
```
\s -> zipWith id (map map [even, (/=0) . flip mod 7]) (repeat s)
```
b)
```
(\a b t -> (a t, b t)) (uncurry (flip const)) (uncurry const))
```
c)
```
let g k 0 = k 1
    g k n = g (k . (n*)) (n - 1)
in f = g id
```
d)
```
f n = foldr (\k -> concat . replicate k) [0] [n,n - 1..1]
```
e)
```
skipping = skip' id where
    {skip' f [x] = [f []]; skip' f (x:xs) = f xs : skip' (f . (x:)) xs}
```
f)
```
let f = 0 : zipWith (+) f g
    g = 1 : zipWith (+) (repeat 2) g}
in f
```
g)
```
boolseqs = []:[b:bs | bs <- boolseqs, b <- [False, True]]
```
h)
```
let f = 1 : 1 : zipWith (+) (tail g) g
    g = 1 : 1 : zipWith (+) (tail f) f
in f
```

Příklad 8.3.6 Napište funkci `find :: [String] -> String -> [String]`, která vrátí všechny řetězce ze seznamu v prvním argumentu, které jsou podřetězcem řetězce v druhém argumentu. Předpokládejte, že všechny řetězce v prvním argumentu jsou neprázdné. Příklad:

`find ["a", "b", "c", "d"] "acegi"` $\rightsquigarrow^*$ `["a", "c"]`
`find ["1", "22", "321", "111", "32211"] "32211"` $\rightsquigarrow^*$ `["1", "22", "32211"]`

Příklad 8.3.7 Které z následujících funkcí není možno v Haskellu definovat (uvažujte standard Haskell98)?

a) `f x = x x`
b) `f = \x -> (\y -> x - (\x -> x * x + 2) y)`
c) `f k = tail . tail . ... . tail` (funkce se opakuje `k`-krát)
d) `f k = head . head . ... . head` (funkce se opakuje `k`-krát)

Příklad 8.3.8 Mějme dány výrazy `x, y`, které mají kompatibilní typ (lze je tedy unifikovat). Navrhněte nový výraz, který bez použití explicitního otypování vynutí, aby `x, y` měli stejný typ. Zkuste najít více řešení.

Příklad 8.3.9 Doplňte do výrazu `foldr f [1,1] (replicate 8 ())` vhodnou funkci místo `f` tak, aby se tento výraz vyhodnotil na prvních deset Fibonacciho čísel v klesajícím pořadí.

Příklad 8.3.10 Určete typ funkce `unfold` definované níže.

```
unfold p h t x = if p x then [] else h x : unfold p h t (t x)
```

Tato funkce intuitivně funguje jako seznamový *anamorfismus* (opak seznamového *katamorfismu* `foldr`). Tedy zatímco katamorfismus zpracovává prvky seznamu a vygeneruje hodnotu, anamorfismus na základě několika daných hodnot seznam vytváří.

Pomocí funkce `unfold` definujte následující funkce:

a) `map`
b) `filter`
c) `foldr`
d) `iterate`
e) `repeat`
f) `replicate`
g) `take`
h) `list_id` (tj. `id :: [a] -> [a]`)
i) `enumFrom`
j) `enumFromTo`

Nejvíce vnější funkcí by měla být funkce `unfold`.

Příklad 8.3.11 Popište množinu funkcí $\{dot_k \mid k \geq 1\}$, kde $dot_k =$ `(.) (.) ... (.)` (k-krát).

Pomůcka: Při zjišťování vlastností funkce dot_k je možné využít zjištěné vlastnosti funkce dot_{k-1}.

Příklad 8.3.12 Je možné jen s pomocí funkce `id` a aplikace vytvořit nekonečně mnoho různých funkcí? A pomocí funkce `const`? (Připomínáme, že skládání je funkce a nemůžete ji tedy použít!)

Příklad 8.3.13 Najděte všechny plně definované konečné seznamy `s` (tj. pro každý jejich prvek platí, že jej lze v konečném čase vyhodnotit), pro které platí:

$\forall$ `f :: t -> t`. $\forall$ `p :: t -> Bool`. `filter p (map f s)` $\equiv$ `map f (filter p s)`

Uvažujte, že `f`, `p` jsou plně definované pro každý argument.

Řešení

Řešení 8.1.1 Nejprve je třeba přemyslet si, jak by taková funkce intuitivně měla fungovat. Katamorfismus je obecně funkce na struktuře, která nahrazuje konstruktory této struktury funkcemi, a ve výsledku umožní vyhodnocení nebo její „kolaps“ na jedinou hodnotu. Datový typ `Nat` má konstruktory `Zero :: Nat` a `Succ :: Nat -> Nat`. Našim cílem je převod hodnoty tohoto typu na nějakou hodnotu, obecně typu `a`. Katamorfismus na hodnotách daného typu je definován funkcemi, které nahrazují jeho datové konstruktory funkcemi stejné arity, jejichž výsledná hodnota je typu `a`, a v místě, kde má konstruktor argument původního typu (v tomto případě tedy `Nat`), uvedeme `a`.

S těmito znalostmi se tedy podívejme na typ `Nat`. `Zero` nahradíme nulární funkcí, tedy hodnotou typu `a`. `Succ` zas nahradíme unární funkcí s typem `a -> a`. Když definujeme tuto transformaci jako funkci, musíme ji definovat po částech pro jednotlivé datové konstruktory. V těle pak použijeme dodané funkce a rekurzivně voláme `natfold`:

```
natfold :: (a -> a) -> a -> Nat -> a
natfold s z Zero     = z
natfold s z (Succ x) = s (nfold s z x)
```

Pokud bychom fixovali parametry `s` a `z`, lze lépe vidět, jak katamorfismus na `Nat` pracuje:

```
natfoldsz :: Nat -> a
natfoldsz Zero     = z
natfoldsz (Succ x) = s (nfoldsz x)
```

Řešení 8.1.2

a)
```
treeSum :: Num a => BinTree a -> a
treeSum = treeFold (\v l r -> v + l + r) 0
```

b)
```
treeProduct :: Num a => BinTree a -> a
treeProduct = treeFold (\v l r -> v * l * r) 1
```

c)
```
treeOr :: BinTree Bool -> Bool
treeOr = treeFold (\v l r -> v || l || r) False
```

d)
```
treeSize :: BinTree a -> Int
treeSize = treeFold (\_ l r -> 1 + l + r) 0
```

e)
```
treeHeight :: BinTree a -> Int
treeHeight = treeFold (\_ l r -> 1 + max l r) 0
```

f)
```
treeList :: BinTree a -> [a]
treeList = treeFold (\v l r -> l ++ [v] ++ r) []
```

g)
```
treeConcat :: BinTree [a] -> [a]
treeConcat = treeFold (\v l r -> l ++ v ++ r) []
```

h)
```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax = treeFold (\v l r -> maximum [v,l,r]) minBound
```

i)
```
treeFlip :: BinTree a -> BinTree a
treeFlip = treeFold (\v l r -> Node v r l) Empty
```

j)
```
treeId :: BinTree a -> BinTree a
treeId = treeFold (\v l r -> Node v l r) Empty
treeId' = treeFold Node Empty
```

k)
```
rightMostBranch :: BinTree a -> [a]
rightMostBranch = treeFold (\v l r -> v:r) []
```

l)
```
treeRoot :: BinTree a -> a
treeRoot = treeFold (\v l r -> v) undefined
treeRoot' = treeFold (const . const) undefined
```

m)
```
treeNull :: BinTree a -> Bool
treeNull = treeFold (\v l r -> False) True
```

n)
```
leavesCount :: BinTree a -> Int
leavesCount = treeFold (\v l r -> if l + r == 0 then 1 else l + r) 0
```

o)
```
leavesList :: BinTree a -> [a]
leavesList = treeFold (\v l r -> if null l && null r then [v]
                                                     else l ++ r) []
```

p)
```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap f = treeFold (\v l r -> Node (f v) l r) Empty
treeMap' f = treeFold (\v -> Node (f v)) Empty
treeMap'' f = treeFold (Node . f) Empty
```

q)
```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny p = treeFold (\v l r -> p v || l || r) False
treeAny' p = treeFold (\v l r -> or [p v, l, r]) False
```

r)
```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair = treeFold (\(x,y) l r -> x == y && l && r) True
```

s)
```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums = treeFold (\v l r -> Node (v + root l + root r) l r) Empty
              where root (Node v l r) = v
                    root Empty = 0
```

t)
```
findPredicates :: a -> BinTree (Int, a -> Bool) -> [Int]
findPredicates x = treeFold (\(n,v) l r -> if v x then l ++ [n] ++ r
                                                  else l ++ r) []
```

Řešení 8.1.3

```
ntreeSize :: NTree a -> Integer
ntreeSize = ntreeFold (\_ szs -> 1 + sum szs)

ntreeMax :: Ord a => NTree a -> a
ntreeMax = ntreeFold (\v vs -> maximum (v:vs))

ntreeDepth :: NTree a -> Integer
```

```
ntreeDepth = ntreeFold (\_ ms -> 1 + maximum (0:ms))

ntreeMap :: (a -> b) -> NTree a -> NTree b
ntreeMap f = ntreeFold (\v ts -> NNode (f v) ts)
```

Řešení 8.2.1

a) `head [id, not]`

Nejprve určíme typy základních podvýrazů (musíme dát pozor, aby typové proměnné v různých podvýrazech byly různé):

```
id :: a -> a
not :: Bool -> Bool
head :: [b] -> b
```

Dále si všimneme, že `id` a `not` jsou oba prvky stejného seznamu, a tudíž musí mít stejný typ. Unifikujeme:
`a -> a` $\sim$ `Bool -> Bool`
Protože typ `Bool` je méně obecný, dostáváme `a = Bool`, a tedy `[id, not] :: [Bool -> Bool]` podle typu prvků seznamu (`Bool -> Bool`).

Dále aplikujeme funkci `head` na seznam, proto musíme unifikovat typ prvního parametru v typu `head` s typem seznamu:
`[b]` $\sim$ `[Bool -> Bool]` a z toho `b` $\sim$ `Bool -> Bool`

Při aplikaci funkce na jeden parametr dojde k odstranění typu prvního parametru z typu funkce a zároveň k dosazení za všechny typové proměnné v prvním parametru zmíněné, celkově tak dostáváme typ
`head [id, not] :: Bool -> Bool`
Vidíme, že výsledek je ve skutečnosti funkcí, a lze jej tedy dále aplikovat, například:
`head [id, not] True` $\rightsquigarrow^*$ `True`

b) `\f -> f 42`

Při typování funkcí (lambda i pojmenovaných) musíme nejprve odvodit typy parametrů a výrazů na levé straně podle vzorů, v nichž jsou použity. Následně se podíváme na pravou stranu definice, odvodíme návratový typ a při tom typicky zpřesníme typy parametrů podle jejich použití.

Pokud se na levé straně nachází proměnná, pak její typ je nějaká dosud nepoužitá polymorfní proměnná, proto odvodíme na začátku
`f :: a`

Nyní se díváme na výrazy na pravé straně a otypujeme nejprve `42 :: Num b => b`. Každá celočíselná konstanta je sama o sobě tohoto typu a ten se může zpřesnit podle místa použití.

Hodnota `f` je aplikovaná na jeden parametr, z čehož odvodíme, že se musí jednat o funkci, a tedy zkonkrétníme typ na `f :: c -> d` (dostáváme unifikaci `a` $\sim$ `c -> d`), což je nejobecnější možný typ (unární) funkce.

`f` je však aplikovaná na `42`, z čehož dostáváme unifikaci
`c` $\sim$ `Num b => b`
a tedy po dosazení zpět do typu pro `f` tento typ:
`f :: Num b => b -> d`
(dosadili jsme `b` za `c`, protože `b` má typový kontext, a tedy je méně obecné).

Typ výrazu `f 42` pak dostáváme odtržením typu prvního parametru z typu `f`:
`f 42 :: d`
Typ celé pravé strany je tedy `d`, což bude i návratový typ.

Funkce má jediný parametr `f :: Num b => b -> d`, celkově tedy dostáváme typ:
`(\f -> f 42) :: Num b => (b -> d) -> d`.

Poznámka: Typový kontext se musí objevit vždy na začátku funkce, proto všechny typové kontexty sloučíme a dáme na začátek.

c) `\t x -> x + x > t x`

Funkce má dva parametry. Jejich vzory jsou proměnné, tedy neomezují jejich typ, proto dostáváme:
`t :: a`
`x :: b`

Doplníme závorky podle priorit operátorů `(+)`, `(>)`:
`\t x -> (x + x) > t x`

Na pravé straně máme použity následující základní výrazy:
`(+) :: Num c => c -> c -> c`
`(>) :: Ord d => d -> d -> Bool`

Z použití `x` jako parametru v `(+)` odvodíme
`b` $\sim$ `Num c => c`
a typ podvýrazu
`x + x :: Num c => c`

Vidíme, že `f` je funkce, tedy zkonkrétníme typ:
`a` $\sim$ `e -> f`, tedy `t :: e -> f`

Zároveň však vidíme, že první argument `t` je `x :: Num c => c`, a tedy zkonkrétníme typ dále se substitucí `e` $\sim$ `Num c => c`:
`t :: Num c => c -> f`.

Nyní se zaměříme na parametry operátoru `(>)`. Levý parametr je `x + x :: Num c => c` a pravý `t x :: f` (z typu `t`). Z typu `(>)` dostáváme unifikace:
`Ord d => d` $\sim$ `Num c => c`
`Ord d => d` $\sim$ `f`
Dostáváme tedy `d` $\sim$ `f` $\sim$ `c`, ale musíme sloučit kontexty, proto:
`t :: (Ord d, Num d) => d -> d`
`x :: (Ord d, Num d) => d`

Typ celého výrazu na pravé straně je pak:
`(x + x) > f x :: Bool` (z návratového typu `(>)`).
Pro typy parametrů vezmeme nejkonkrétnější odvozené typy (ty co jsme viděli naposledy),

celkově dostáváme typ:
`(\f x -> x + x > f x) :: (Ord d, Num d) => (d -> d) -> d -> Bool`

Poznámka: Typový kontext zde nelze zjednodušit, protože `Ord` není nadtřídou `Num` ani naopak.

d) `\xs -> filter (> 2) xs`

Začínáme s

```
xs :: a
filter :: (b -> Bool) -> [b] -> [b]
2 :: Num d => d
(>) :: Ord e => e -> e -> Bool
```

Částečnou aplikací `(>)` na `2` dostáváme `Num d => d` $\sim$ `Ord e => e` a
`(> 2) :: (Ord d, Num d) => d -> Bool`.

Dále částečnou aplikací `filter` na `(> 2)` dostáváme unifikaci `b -> Bool` $\sim$ `(Ord d, Num d) => d -> Bool`, a tedy `b` $\sim$ `(Ord d, Num d) => d` a
`filter (> 2) :: (Ord d, Num d) => [d] -> [d]`.

Nyní aplikujeme tuto funkci na argument `xs`, kde dostaneme
`a` $\sim$ `(Ord d, Num d) => [d]`
a celkový typ pravé strany je
`filter (> 2) xs :: (Ord d, Num d) => [d]`.

Jediným parametrem funkce je `xs :: (Ord d, Num d) => [d]`, celkem tedy dostáváme typ
`(\xs -> filter (> 2) xs) :: (Ord d, Num d) => [d] -> [d]`

e) `\f -> map f [1,2,3]`

```
1 :: Num a => a
[1,2,3] :: Num a => [a]
map :: (b -> c) -> [b] -> [c]
f :: d -- parametr
```

Z aplikace `map f` dostáváme `d` $\sim$ `b -> c`, a tedy `f :: b -> c`, `map f :: [b] -> [c]`.

Tuto funkci dále aplikujeme na `[1,2,3]`: `[b]` $\sim$ `Num a => [a]`,
`map f [1,2,3] :: [c]`
`f :: Num a => a -> c`

Celá funkce je tedy typu `(\f -> map f [1,2,3]) :: Num a => (a -> c) -> [c]`.

f)
```
foo f True  = map f [1,2,3]
foo f False = filter f [1,2,3]
```

Začneme s

```
f :: a        -- 1. parametr
True :: Bool  -- 2. parametr
False :: Bool -- dalsi vzor pro 2. parametr, typy unifikovatelne => OK

map :: (b -> c) -> [b] -> [c]
filter :: (d -> Bool) -> [d] -> [d]
```

```
[1,2,3] :: Num a => [a]
```

Dále pokračujeme obdobně jako v předchozím případě, ale pro každý vzor zvlášť:

```
-- z prvniho vzoru:
f :: Num a => a -> c
map f [1,2,3] :: [c]

-- z 2. vzoru:
f :: Num a => a -> Bool
filter f [1,2,3] :: Num a => [a]
```

Unifikujeme oba typy pro `f`:
`c` $\sim$ `Bool`
`f :: Num a => a -> Bool`
a tedy zpřesníme typ `map f [1,2,3] :: [Bool]`

Rovněž i návratová hodnota musí být vždy stejná: `[Bool]` $\sim$ `Num a => [a]`

Z čehož odvodíme, že funkce je otypovatelná pouze za předpokladu, že typ `Bool` je instancí typové třídy `Num`, což není pravda, a tedy funkci `foo` nelze otypovat.

g) `\(p,q) z -> q (tail z) : p (head z)`

```
p :: a
q :: b
(p,q) :: (a,b) -- prvni argument
z :: c         -- druhy argument
tail :: [d] -> [d]
head :: [e] -> e
(:) :: f -> [f] -> [f]
```

Z použití v `tail, head` můžeme odvodit, že `z` je seznam:
`c` $\sim$ `[d]` $\sim$ `[e]`, `z :: [d]`.

Dále tedy: `tail z :: [d]`, `head z :: d`.

Funkce `q` je tedy aplikovaná na parametr typu `[d]`, a proto její typ bude `q :: [d] -> g` (unifikace `b` $\sim$ `[d] -> g`).

Obdobně pro `p`: `p :: d -> h` (unifikace `a` $\sim$ `e -> h` $\sim$ `d -> h`).

Tedy `q (tail z) :: g`, `p (head z) :: h` jsou parametry `(:)`, a tedy dostáváme:
`f` $\sim$ `g`, `[f]` $\sim$ `h` $\Rightarrow$ `[f]` $\sim$ `h` $\sim$ `[g]` a po dosazení:
`q (tail z) :: g`
`p (head z) :: [g]`.

Celý výraz na pravé straně má tedy typ
`q (tail z) : p (head z) :: [g]`.

A celá funkce
`(\(p,q) z -> q (tail z) : p (head z)) :: (d -> [g], [d] -> g) -> [d] -> [g]`.

Řešení 8.2.2

a) V obou řádcích definice lze ze vzoru odvodit, že první parametr je typu `Foo a`. Typová proměnná `a` je tu proto, že zatím nevíme, jaké budou požadavky na typ hodnot uvnitř `Foo` (pozor, samotné `Foo` by bylo špatně, protože to je unární typový konstruktor a jako takový nemůže mít hodnoty).

V prvním řádku je pak `xs :: [a]` (z definice `Bar`), a tedy návratová hodnota je `[a]`.

V druhém řádku je `x :: a` (z definice `Baz`), návratová hodnota je pak `[a]`.

V obou případech jsou návratové hodnoty stejné, tedy nemusíme provádět unifikaci a návratová hodnota celého výrazu je `[a]`.

Celkově dostáváme: `getlist :: Foo a -> [a]`.

b) Na začátku odvodíme `foo :: a`. Díky použití `getlist` však můžeme zpřesnit na `foo :: Foo b` a určit `getlist foo :: [b]` (unifikace $a \sim$ `Foo b`).

Dále potřebujeme znát typy dalších použitých funkcí a výrazů:

```
foldr :: (c -> d -> d) -> d -> [c] -> d
(+)   :: Num n => n -> n -> n
0     :: Num m => m
```

Nyní postupně typujeme aplikaci `foldr`:

```
foldr (+) :: Num n => n -> [n] -> n       -- c ~ d ~ Num n => n
foldr (+) 0 :: Num n => [n] -> n          -- Num m => m ~ Num n => n
foldr (+) 0 (getlist foo) :: Num n => n   -- b ~ Num n => n
```

Kombinací dostáváme typ lambda funkce:

```
(\foo -> foldr (+) 0 (getlist foo)) :: Num n => Foo n -> n
```

Řešení 8.2.3

```
minmax :: Ord a => [a] -> (a, a)
minmax [x]    = (x, x)
minmax (x:xs) = let (mi, ma) = minmax xs in (min x mi, max x ma)

minmax' :: Ord a => [a] -> (a, a)
minmax' (x:xs) = foldr (\x (mi, ma) ->  (min x mi, max x ma)) (x, x) xs

minmaxBounded :: (Ord a, Bounded a) => [a] -> (a, a)
minmaxBounded [] = (maxBound, minBound) -- proc musi byt tohle naopak?
minmaxBounded (x:xs) = let (mi, ma) = minmaxBounded xs in (min x mi, max x ma)

minmaxBounded' :: (Ord a, Bounded a) => [a] -> (a, a)
minmaxBounded' = foldl (\ (mi, ma) x -> (min x mi, max x ma))
                       (maxBound, minBound)
```

Řešení 8.2.4

a)
```
\f -> (map . uncurry) f
\f -> map (uncurry f)
```

```
\f xs -> map (uncurry f) xs

\f xs -> [ f a b | (a, b) <- xs ]
```

b)
```
(\f xs -> zipWith (curry f) xs xs) :: ((a, a) -> b) -> [a] -> [b]
-- funkce je jiz v pointwise

\f xs -> [ f (x, x) | x <- xs ]
```

c)
```
\xs -> (map (* 2) . filter odd . map (* 3) . map (`div` 2)) xs
\xs -> map (* 2) (filter odd (map (* 3) (map (`div` 2) xs)))

\xs -> [ y * 2 | x <- xs, let y = div x 2 * 3, odd y ]
```

d)
```
\xs -> (map (\f -> f 5) . map (+)) xs
\xs -> map (\f -> f 5) (map (+) xs)

\xs -> [ (\f -> f 5) ((+) x) | x <- xs ]
\xs -> [ x + 5 | x <- xs ] -- zjednoduseni (aplikace lambda funkce)
```

Řešení 8.2.5

a)
```
readFile "/etc/passwd" :: IO String
putStrLn "bla" :: IO ()
```

Protože typ výsledku operátoru `(>>)` je stejný jako typ jeho druhého argumentu, typ celého výrazu je `IO ()`.

Přepis do `do`-notace:

```
do readFile "/etc/passwd"
   putStrLn "bla"
```

b)
```
f :: a
putStrLn "bla" :: IO ()
(>>=) :: IO b -> (b -> IO c) -> IO c

-- castecna aplikace (>>=)
(>>=) (putStrLn "bla") :: (() -> IO c) -> IO c

-- a tedy dostavame typ f:
f :: () -> IO c

-- a celeho vyrazu:
(\f -> putStrLn "bla" >>= f) :: (() -> IO c) -> IO c
```

Přepis do `do`-notace:

```
\f -> do x <- putStrLn "bla"
         f x
```

c)
```
getLine >>= \x -> return (read x)
```

```
getLine :: IO String
return :: a -> IO a
read :: Read b => String -> b

-- z typu getLine a read:
x :: String

read x :: Read b => b
return (read x) :: Read b => IO b
getLine >>= \x -> return (read x) :: Read b => IO b
```

Přepis do `do`-notace:

```
do x <- getLine
   return (read x)
```

d) Nelze otypovat. Z `IO` není úniku, druhý parametr `(>>=)` musí vždy vracet `IO` akci (snaha o substituci `Read a => IO a` ∼ `Integer`).

Řešení 8.2.6

a) Nekorektní, funkci `max` možno aplikovat maximálně na dva argumenty.

b) Korektní, `(False < True) || True` ⇝ `True || True` ⇝ `True`. `Bool` je instancí typové třídy `Ord`, a tedy hodnoty tohoto typu lze porovnávat.

c) Nekorektní. Sice `5.1 ^ (3.2 ^ 2)` ⇝ `5.1 ^ 10.24`, ale umocňování `(^)` je typu

```
(^) :: (Num a, Integral b) => a -> b -> a
```

a hodnotu `10.24` není možno otypovat `(Integral b) => b`.
Poznámka: Šlo by použít jiný operátor umocnění. Haskell poskytuje tři a liší se povoleným typem argumentů:

```
(^) :: (Integral b, Num a) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(**) :: Floating a => a -> a -> a
```

d) Korektní, `2 ^ (if (even m) then 1 else m)`. Předpokládá se, že `m` je definováno (a je celočíselné).

e) Nekorektní. První chybou je chybný zápis funkce `mod`. Správně je buď `mod m 2` nebo `m` `` `mod` `` `2`. Další problém tvoří typy:

```
mod :: (Integral a) => a -> a -> a,
(/) :: (Fractional b) => b -> b -> b,
(+) :: (Num c) => c -> c -> c
```

Po aplikaci argumentů na funkce `mod` a `(/)` dostaneme

```
mod m 2 :: (Integral a) => a
11 / m :: (Fractional b) => b
(+) :: (Integral a, Fractional a) => a -> a -> a
```

`Integral` a `Fractional` jsou však nekompatibilní typové třídy, neexistuje tedy typ patřící do obou tříd. Otypování tedy není možné a výraz je nekorektní.
Tento příklad ilustruje fakt, že typový systém může odmítnout i výrazy, které jsou intuitivně korektní. V tomhle případě to můžeme vyřešit následovně:

```
fromIntegral (mod m 2) + 11 / m
```

f) Korektní, `((/) 3 2) + 2`.
g) Nekorektní. Uspořádané dvojice a trojice (ve všeobecnosti libovolné k-tice) jsou vždy navzájem různé typy. Seznam vytvořený v zadání by tedy nebyl homogenní.
h) Nekorektní, abstraktor `\s` je umístěný v prvním prvku uspořádané dvojice, tedy má platnost jenom tam. Výraz by však mohl být korektní, jestli by bylo `s` definováno na vyšší úrovni.
i) Korektní, definovanou lokální proměnnou `m` není nutné použít.
j) Nekorektní, v druhé členu trojice nesedí typy – oba řetězce jsou stejného typu.
k) Korektnost závisí na typech `f` a `x`. Výraz je korektní, pouze když typ `f t` (kde `t` je prvek seznamu `x`) je kompatibilní s typem `[a]`.
l) Korektní, vytvoří jednoprvkový seznam.
m) Korektní, ekvivalentní prázdnému seznamu: `fst (map, 3) fst []` ⇝ `map fst []` ⇝ `[]`.
n) Nekorektní, zápis `[`a`,`b`..`c`]` je možné použít jenom u typů nacházejících se v typové třídě `Enum`.
o) Nekorektní, výraz `(x:_)` je vzor, který není možno použít na místech pro výraz. Vzory lze použít pouze v λ-abstrakci, jako argumenty při definici funkce a v konstrukcích `case`, `where` nebo nalevo od `<-`.
p) Nekorektní, funkce `fst` operuje pouze na uspořádaných dvojicích, ne na seznamech.
q) Korektní, v generátoru lze použít na levém místě vzor, tedy i `_`. V tomto případě je výsledkem prázdný seznam, protože generátor neposkytne žádný prvek.
r) Nekorektní, syntax intensionálních seznamů je `[ expr | rule, ..., rule ]`. Nemůžeme uvést svislítko a část s pravidly dvakrát.
s) Nekorektní. Zápis u druhého seznamu není možný. Před `..` lze uvést nejvíce dvě hodnoty – počáteční a druhou (pro určení diference). Navíc před dvěma tečkami se nikdy neuvádí čárka.
t) Korektní. `getLine` vrací vnitřní hodnotu typu `String`, kterou pomocí operátoru `>>=` dáme jako argument funkci `putStrLn`.
u) Nekorektní. Vnitřní hodnotu `getLine` zahodíme, ale za `>>` musí být výraz typu `IO a`, avšak `putStrLn :: String -> IO ()`, a tedy tyto dva typy nejsou kompatibilní.
v) Korektní. Výsledek `getLine` se zahodí a vypíše se řetězec získaný z λ-abstrakce.
w) Korektní, jde o funkci, která bere jako vstup řetězcový argument a na výstup vypíše nejprve `OK` a následně zadaný řetězec. Následovné výrazy jsou totiž ekvivalentní:

```
(>>) (putStrLn "OK") . putStrLn
\x -> (>>) (putStrLn "OK") (putStrLn x)
putStrLn "OK" >> putStrLn x
```

x) Nekorektní, `f >>= g :: IO a`, avšak celý tento výraz je argumentem prvního `>>=` a musí mít tedy typ kompatibilní s typem `b -> IO c`, což neplatí.

Řešení 8.2.7

a) Nekorektní, správně je buď `(Int, Int)`, tj. uspořádaná dvojice tvořená dvěma hodnotami typu `Int`, anebo `[Int]`, tj. seznam hodnot typu `Int`.
b) Korektní, ekvivalentní s typem `Int`.
c) Nekorektní, zapsaný výraz je ekvivalentní s `[()] -> []`, avšak `[]` je pouze unární typový konstruktor (ne typ) a vždy musí „obalovat“ nějaký typ.
d) Korektní.

e) Korektní.
f) Nekorektní, typové proměnné musí začínat malým písmenem, jinak jde o typový konstruktor, avšak `A` není standardním typovým konstruktorem.
g) Nekorektní, unární typový konstruktor `[]` nemá argument.
h) Korektní, `IO` je běžný unární typový konstruktor. Výrazem s kompatibilním typem je například `return putStrLn`.
i) Korektní, například `return (Just 1)` má kompatibilní typ.
j) Korektní.
k) Korektní, všechny použité názvy jsou typové proměnné (nemůžou to být obyčejné typy, jelikož začínají malým písmenem). Tento typ je ekvivalentní typu `[a] -> b -> c -> a`
l) Nekorektní, typový kontext musí odkazovat na typovou proměnnou, která je použitá. Tedy například když při určování typu výrazu vypadne typová proměnná, na kterou je navázána typová třída, je potřeba toto omezení z typového kontextu vypustit.
m) Nekorektní, typový kontext musí být vždy umístěn na začátku typu.
n) Korektní, sice každý typ v typové třídě `Integral` je i v typové třídě `Num`, a tedy uvedení `Num` je zbytečné, není to chybou. Jenom je potřeba myslet na to, že pokud typový kontext obsahuje více omezení, musí být umístěny v závorkách.
o) Nekorektní, typový kontext nelze takhle zkracovat. Musí být vždy uveden na začátku typu, tj. `Num a => a -> a`.
p) Nekorektní, typovým kontextem nelze nahradit typovou proměnnou, správně by mohlo být třeba `Num a => a -> c -> c`.

Řešení 8.2.8

a) `[2]`
b) Dojde k chybě vyhodnocování – `head`, `tail` nejsou definovány na prázdném seznamu.
c) `[]`
d) `[]`
e) `[map (0:) []]` ⇝ `[[]]`
f) `[(++[]) [[], []], (++[]) [], (++[]) [[]]]` ⇝
   ⇝ `[[[], []], [], [[]]]`
g) `[]`
h) `[[]]`
i) `[]`
j) `3 * 5 + (\x -> x + x ^ 2) (2 * 5 - 1)` ⇝
   ⇝ `15 + (2 * 5 - 1) + (2 * 5 - 1) ^ 2` ⇝* `15 + 9 + 9 ^ 2` ⇝ `105`
k) `(.) id (max 5) 3` ⇝ `id (max 5 3)` ⇝* `5`
l) `map f (x ++ [])` ⇝ `map f x`

Řešení 8.2.9

a) Ne, první výraz se snaží aplikovat funkci `(+1)` na `(*2)`, což není číslo. Druhý je ekvivalentní s výrazem `\x -> (+1) ((*2) x)`, a tedy s `\x -> x * 2 + 1`.
b) Ne, správně má být `f . (.g)` ⇝ `\x -> f ((.g) x)` ⇝ `\x -> f (x . g)`
c) Ne, tělo λ-abstrakcí se táhne tak daleko doprava, jak je to možné (v tomhle případě je to možné až po úplný konec výrazu). Implicitní uzávorkování je následovné:

```
getLine >>= (\x -> (putStrLn (reverse x)) >> (putStrLn "done"))
```

d) Ne, ve všeobecnosti není možné dělat „lifting“ λ-abstrakce tímto způsobem. První výraz byl nekorektní (definice `(+)` neumožňuje sečíst funkci a hodnotu), zatímco druhý výraz být korektní může.
e) Ne, operandy operátoru `&&` se vyhodnocují zleva. V případě, když `s1 == s2` a oba seznamy jsou nekonečné, vyhodnocování levého argumentu `&&` nikdy neskončí. Nedojde tedy ke zjednodušení celého výrazu na `False` i když druhým argumentem je `False`.

Řešení 8.2.10

a) `[a]`
b) `[()]`
c) `[Bool]`, tady pozor na to, že výsledkem je sice `[]` a ten má například po otypování v interpretu typ `[a]`, avšak tento prázdný seznam vznikl ze seznamu typu `[Bool]` a této stopy se už nelze zbavit.
d) `(a -> c) -> a -> c`
e) `b -> (b -> c) -> c`
f) `a -> a`
g) `(a -> b -> c) -> a -> (d -> b) -> d -> c`
h) `[a -> b -> c] -> [b -> a -> c]`
i) `(b -> b1 -> c) -> (a -> b) -> a -> (a1 -> b1) -> a1 -> c`
j) `((a -> [a] -> [a]) -> [a1] -> t) -> t`
k) `[[Bool]] -> [[Bool]]`
l) Nesprávně utvořený výraz – nemožno sestrojit nekonečný typ.
m) `[t] -> [a]`
n) `IO a -> IO String`
o) `IO ()`, typem u `do`-konstrukcí je vždy typ posledního výrazu/akce.

Řešení 8.2.11 Funkce vybere nultý prvek, zahodí $k-1$ prvků a rekurzivně se zavolá.

```
nth1 :: Int -> [a] -> [a]
nth1 _ []    = []
nth1 k (x:s) = x : nth1 k (drop (k - 1) s)
```

Funkce si udržuje index prvku i (modulo k) a jestli je rovný 0, prvek použije, jinak ho zahodí.

```
nth2 :: Int -> [a] -> [a]
nth2 k s = nth2' k 0 s where
    nth2' k i (x:s) = (if i==0 then (x:) else id) $ nth2' k (mod (i+1) k) s
    nth2' _ _ []    = []
```

Nejdříve „slepí“ seznam se seznamem `[0..]` a vybere pouze ty prvky, které jsou připojené k násobkům čísla k.

```
nth3 :: Int -> [a] -> [a]
nth3 k s = map fst $ filter ((==0) . flip mod k . snd) $ zip s [0..]
```

Vytvoří seznam seznamů po k prvcích a následně z nich vybere pouze první prvky.

```
nth4 :: Int -> [a] -> [a]
nth4 k s = map head $ nth4' k s where
    nth4' _ [] = []
    nth4' k s  = take k s : nth4' k (drop k s)
```

Řešení 8.2.12

```
modpwr :: Integer -> Integer -> Integer
modpwr _ 0 _ = 1
modpwr n k m = mod (if even k then t else n * t) m
    where t = modpwr (mod (n^2) m) (div k 2) m
```

Méně efektivní řešení s lineární časovou složitostí by mohlo vypadat takhle:

```
modpwr' :: Integer -> Integer -> Integer
modpwr' _ 0 _ = 1
modpwr' n k m = mod (n * modpwr' n (k-1) m) m
```

Řešení 8.2.13

a) Funkce `f1` bere jako argument funkci, která dostane argument 0.

```
f1 :: Num b => (b -> c) -> c
f1 x ⇝ flip id 0 x ⇝ id x 0 ⇝ x 0
```

b) Funkce `f2` vloží svůj argument do seznamu.

```
f2 :: a -> [a]
f2 x ⇝ flip (:) [] x ⇝ (:) x [] ≡ x:[] ≡ [x]
```

c) Funkce `f3` vezme dva seznamy a vrátí první seznam zkrácený na délku kratšího z nich.

```
f3 :: [a] -> [b] -> [a]
f3 [1,2,3] [4,5] ⇝ zipWith const [1,2,3] [4,5] ⇝* [const 1 4, const 2
   5] ⇝* [1,2]
```

d) Funkce `f4` vezme hodnotu typu `Bool` a vrací funkci, která pro `True` vrátí svůj argument s předřetězeným `'/'` a pro `False` vrátí původní argument beze změny.

```
f4 :: Bool -> [Char] -> [Char]
f4 True "yes" ⇝ (if True then ('/':) else id) "yes" ⇝ ('/':) "yes" ≡
   "/yes"
```

e) Funkce `f5` postupně odzadu aplikuje funkce ze seznamu v prvním argumentu na hodnotu v druhém argumentu.

```
f5 :: Num b => [b -> b] -> b
f5 [(3*),(+7)] ⇝ foldr id 0 [(3*),(+7)] ⇝* id (3*) (id (+7) 0) ⇝* (3*)
   ((+7) 0) ≡ 3 * (0 + 7)
```

f) Funkce `f6` je ekvivalentní s funkcí `foldr (\x s -> not s) True`. Hodnota prvků v seznamu se tedy vůbec nevyužívá, avšak za každý prvek se vygeneruje jedno `not` a všechny se postupně aplikují na `True`. Funkce tedy zjišťuje, jestli má seznam sudou délku.

```
f6 :: [a] -> Bool
f6 [1,2,3] ⇝* not (not (not True)) ⇝ False
```

Řešení 8.2.14

a) Platí, ale jenom jestli je `s` konečné.

b) Neplatí.

```
map f . filter (p . f) ≡ filter p . map f
```

c) Neplatí kvůli speciálnějšímu typu `flip . flip`. Pokud bychom nebrali typy do úvahy, platilo by.
`flip . flip` ≡ `(id :: (a -> b -> c) -> (a -> b -> c))`
d) Neplatí.
`foldr f (foldr f z s) t` ≡ `foldr f z (t ++ s)`
e) Neplatí, seznamy mohou být různé délky nebo některý může být nekonečný.
f) Neplatí pro prázdný seznam.
g) Platí.
h) Platí.

Řešení 8.2.15

a) `\x y -> f (g x) y` ≡ `f . g`
b) `\x y -> f (g (h1 x) (h2 y))`
c) `\x y -> if p y then f x y else x`
d) Doplnění není možno provést, protože čtvrtý argument funkce `foldr` není seznam, ale uspořádaná dvojice.
e) `\x y -> y` ≡ `flip const`

Řešení 8.2.16

a) Konstantní.
b) Lineární.
c) Lineární.
d) Konstantní.
e) Lineární k minimu hodnot `m`, `n`.
f) Lineární k délce seznamu `m`.
g) Výpočet nekončí.
h) Konstantní.
i) Konstantní.

Řešení 8.3.1

```
hanoi :: Int -> Int -> Int -> [(Int,Int)]
hanoi 1 source dest = [(source, dest)]
hanoi n source dest = (hanoi (n-1) source (6 - source - dest)) ++
                      (hanoi 1 source dest) ++
                      (hanoi (n-1) (6 - source - dest) dest)
```

Řešení 8.3.2 Budeme postupovat matematickou indukcí. Hledaným tvrzením je, že tyto funkce jsou ekvivalentní `($)`. Báze indukce je zřejmá. Nechť $\texttt{dollar}_n$ je funkce s n výskyty `($)`. Předpokládejme, že $\texttt{dollar}_n \equiv$ `($)`. Pak $\texttt{dollar}_{(n+1)} \equiv$ `($) . dollar` $_n \equiv$ `($) . ($)` a stačí tedy dokázat `($) . ($)` ≡ `($)`. Máme

`($) . ($)` ≡ `\x -> ($) (($) x)` ≡ `\x y -> (($) x) $ y` ≡ `\x y -> (($) x) y` ≡ `($)`.

Intuitivně, operátor `($)` funguje jako identita na unárních funkcích, a tedy složení identit je logicky opět identita.

Řešení 8.3.3

```
if' :: Bool -> a -> a -> a
if' cond th el = [el, th] !! fromEnum cond
```

nebo také

```
if' :: Bool -> a -> a -> a
if' True  t f = t
if' False t f = f
```

Řešení 8.3.4

```
head $ [ val1 | cond1 ] ++ [ val2 | cond2 ] ++ ... ++ [ val_default ]
```

Řešení 8.3.5

a) Lze nahlédnout, že ve výrazu `zipWith id x y` musí být `x` a `y` seznamy. Sémantiku lze nejsnáze přiblížit příkladem:

`zipWith id [f, g, h] [x, y, z]` ⇝ `[f x, g y, h z]`

První seznamový argument lze upravit následovně:

```
   map map [even, (/=0) . flip mod 7]
⇝ [map even, map ((/=0) . flip mod 7)]
```

Protože tento seznam má dva prvky a v druhém seznamovém argumentu je použit `repeat`, výsledek aplikace `zipWith` bude mít vždy dva prvky. Tedy výraz ze zadání můžeme na základě těchto znalostí upravit na následovný ekvivalentní výraz:

`\s -> [map even s, map (/=0) . flip mod 7) s]`

Teď vidíme, že vzhledem na fakt `even :: Integral a => a -> Bool` a `mod :: Integral a => a -> a -> a` je typ našeho výrazu `Integral a => [a] -> [[Bool]]`. Slovně popsáno, náš výraz vrací pro seznam celých čísel seznam, kterého prvním prvkem je seznam indikující paritu vstupního seznamu a druhým prvkem je seznam indikující jestli dávají prvky vstupního seznamu nenulový zbytek po dělení sedmi:

```
f [1..10] ⇝* [[False,True,False,True,False,True,False,True,False,True],
    [True,True,True,True,True,True,False,True,True,True]]
```

b) Výraz představuje funkci, která se v knihovně nazývá `swap`.

`uncurry (flip const)` ≡ `snd`, `uncurry const` ≡ `fst`

```
\t -> (snd t, fst t)
\(a, b) -> (b, a)
```

c) Funkce `f` vrací faktoriál svého argumentu: První argument funkce `g` slouží jako akumulátor, do kterého se postupně vytváří složení funkcí `(n*)` pro všechny `n` mezi 1 a hodnotou argumentu funkce `f`. Na závěr se tato násobící funkce aplikuje na jedničku:

```
f 0 ⇝ g id 0 ⇝ id 1 ⇝ 1
f 1 ⇝ g id 1 ⇝ g (id . (1*)) 0 ⇝ (id . (*1)) 1 ⇝* 1 * 1
f 2 ⇝ g id 2 ⇝ g (id . (2*)) 1 ⇝ g (id . (2*) . (1*)) 0 ⇝* 2 * 1 * 1
f 3 ⇝ g id 3 ⇝ g (id . (3*)) 2 ⇝ g (id . (3*) . (2*)) 1 ⇝
    g (id . (3*) . (2*) . (1*)) 0 ⇝* 3 * 2 * 1 * 1
...
```

d) `\k -> concat . replicate k` ≡ `\k x -> concat (replicate k x)` Označme si první argument funkce `foldr` pro snazší manipulaci jako `corep`. Při vyhodnocování budeme pro lepší názornost postupovat striktní vyhodnocovací strategií, tj. od nejvnitřnějších volání funkcí. Pak dostáváme následovné výrazy:

```
f 0 ⇝ [0]
f 1 ⇝ foldr corep [0] [1] ⇝* [0]
f 2 ⇝ foldr corep [0] [2,1] ⇝ corep 2 (foldr corep [0] [1]) ⇝*
    corep 2 [0] ⇝* [0,0]
f 3 ⇝ foldr corep [0] [3,2,1] ⇝ corep 3 (foldr corep [0,0] [2]) ⇝*
    corep 3 [0,0] ⇝* [0,0,0,0,0,0]
...
```

Na základě fungování funkce si lze všimnout, že `f` generuje pro argument `n` seznam s `n!` prvky `0`.

e) `skipping [1,2,3,4]` ⇝ `[[2,3,4], [1,3,4], [1,2,4], [1,2,3]]`

f) Nejprve se podíváme na funkci `g`, jelikož nezávisí na `f`.

```
g = [1,3..]
```

Následně

```
f = 0 : zipWith (+) f [1,3..]
f !! n == g !! (n - 1) + f !! (n - 1) == 2 * n - 1 + f !! (n - 1)
```

Odsud se pomocí indukce dá dokázat `f == map (^2) [0..]`, tedy `f` je seznam druhých mocnin nezáporných celých čísel.

g) Prohledáváním BFS (Breadth-first search) generuje všechny konečné seznamy typu `[Bool]`:

```
[[], [False], [True], [False, False], [True, False], ...]
```

přičemž ke každému seznamu vytvoří dva nové tak, že na začátek jednoho připojí `False` a na začátek druhého `True`. Seznam generuje jako frontu.

h) Definice funkcí `f` a `g` jsou až na záměnu `f` a `g` totožné. Celý výraz tedy můžeme přepsat do tvaru `f = 1 : 1 : zipWith (+) (tail f) f in f`, což představuje seznam Fibonacciho čísel.

Řešení 8.3.6

```
find l s = [p | p <- l, or (map (\x -> p == zipWith const x p) (suffixes s))]
  where
    suffixes "" = []
    suffixes (x:s) = (x:s) : suffixes s
```

Řešení 8.3.7

a) Nelze, funkce není otypovatelná. Argument `x` musí mít nějaký typ, řekněme `a`. Z pravé strany dostáváme specializaci `x` ≡ `a1 -> a2`. Funkci typu `a1 -> a2` však můžeme aplikovat pouze na argument typu `a1`. Avšak jejím argumentem je opět `x` s typem `a1 -> a2`. Dostáváme tedy `a1` ≡ `a1 -> a2` což představuje tzv. *nekonečný typ*. Tato funkce tedy není otypovatelná, potažmo jí nelze definovat.

b) Lze, výraz je možné přepsat do tvaru `\x y -> x - (\x -> x * x + 2) y`. Překrývání formálního argumentu `x` ve vnitřní λ-abstrakci je povolené – hodnota `x` v součinu bude daná nejvnitřnější (nejbližší) λ-abstrakcí.

c) Lze, funkci je možné zapsat jako `f k = foldr (.) id (replicate k tail)` nebo taky `f = drop`

d) Nelze, funkce `f` musí mít jednoznačný typ. Pro funkci v zadání by platilo

```
f 1 :: [a] -> a
f 2 :: [[a]] -> a
f 3 :: [[[a]]] -> a
```

což jsou nekompatibilní typy.

Řešení 8.3.8

```
unused1 = [x, y]
unused2 = if True then x else y
unused3 1 = x; unused3 2 = y
unused4 = y `asTypeOf` x
```

Řešení 8.3.9 `f = \_ (x:y:s) -> (x + y) : (x:y:s)`

Řešení 8.3.10 Parametr `p` představuje zastavovací podmínku, parametr `h` modifikaci prvků předávaných „na výstup“, parametr `t` modifikaci seznamu, na kterém se bude dále pracovat, a parametr `x` iniciální hodnotu řídící běh `unfold`. Celkový typ je následovný:

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
```

a) `map f = unfold null (f . head) tail`

b) Nelze tak, aby funkce `unfold` byla nejvíce vnější funkcí.

```
filter p s = unfold null head (dropWhile (not . even) . tail)
    . dropWhile (not . even)
```

c) Není možno, výstupem `unfold` je vždy seznam.

d) `iterate = unfold (const False) id`

e) `repeat = unfold (const False) id id`

f) `replicate n x = ((==0) . fst) snd (\(n,x) -> (pred n,x)) (n,x)`

g)
```
take n x = unfold ((==0) . fst) (head . snd)
      (\(n,s) -> (n-1, tail s)) (n,x)
```

h) `list_id = unfold null head tail`

i) `enumFrom = unfold (const False) head succ`

j) `enumFromTo m n = unfold (>n) head succ m`

Řešení 8.3.11 Množina je tvořena funkcemi $dot_1, dot_2, \ldots, dot_9$ a platí $dot_{n+4} \equiv dot_n$ pro $n \geq 6$.

```
dot_1 = \a b c -> a (b c)
dot_2 = \a b c d -> a b (c d)
dot_3 = \a b c d -> a (b c d)
dot_4 = \a b c d e -> a b c (d e)
dot_5 = \a b c d -> a (b (c d))
dot_6 = \a b c d e -> a (b c) (d e)
dot_7 = \a b c d e -> a b (c d e)
```

```
dot_8 = \a b c d e -> a (b c d e)
dot_9 = \a b c d e f -> a b c d (e f)
```

Řešení 8.3.12 Pouze pomocí `id` to možné není. Každý výskyt `id` se může přepsat podle definice na hodnotu jejího argumentu a jelikož jediná použitá hodnota je `id`, výsledkem je vždy funkce `id`.

Pomocí funkce `const` to však je možné udělat. Například funkce

```
const
const const
const (const const)
const (const (const const))
...
```

jsou ekvivalentní funkcím

```
\x1 x2 -> x1
\x1 x2 x3 -> x2
\x1 x2 x3 x4 -> x3
\x1 x2 x3 x4 x5 -> x4
...
```

což jsou zjevně navzájem různé funkce a je jich nekonečně mnoho.

Řešení 8.3.13 Uvažujme, že řešením jsou seznamy `s` typu `s :: [t]`. Nechť `|t|` představuje počet plně definovaných hodnot typu `t`. Rozeberme možnosti:

- `|t|` $= 0$
  Vzhledem na to, že neexistuje žádná plně definovaná hodnota tohoto typu, jediným seznamem, který za těchto podmínek bereme do úvahy, je `[]` a pro něj platí zadaná podmínka triviálně.
- `|t|` $= 1$
  Nutně `f = id`, odkud `map f = map id = id`, tedy podmínka platí po každý seznam typu `[t]`.
- `|t|` > 1 Rozeberme několik případů podle obsahu seznamu `s`:
  - `s = []`
    Podmínka platí triviálně.
  - `s` obsahuje pouze prvky `a :: t`
    Nechť `b` je hodnota typu `t`, jiná než `a`. Uvažme dále funkce `f = const b` a `p = (b /=)`. Pro tento výběr podmínka neplatí, protože `filter p (map f s) = []` $\not\equiv$ `map f (filter p s) = map f s`.
  - `s` obsahuje alespoň dva různé prvky `a, b :: t`
    Uvažme funkce `f = const a` a `p = (a/=)`. Pro tento výběr podmínka neplatí, protože `filter p (map f s) = []` $\not\equiv$ `map f (filter p s)`

Závěr: Dané tvrzení platí pouze pro prázdné seznamy libovolného typu a pro libovolné seznamy typu `[()]` nebo typu izomorfního.