Cvičení 7

7.1 N-ární stromy

Příklad 7.1.1 Uvažte typ n-árních stromů definovaný následovně:

```
data NTree a = NNode a [NTree a]
             deriving (Show, Read)
```

Definujte následující:

a) funkci `ntreeSize :: NTree a -> Integer`, která spočítá počet uzlů ve stromě
b) funkci `ntreeSum :: Num a => NTree a -> a`, která sečte ohodnocení všech uzlů stromu
c) funkci `ntreeMap :: (a -> b) -> NTree a -> NTree b`, která bere funkci a strom, a aplikuje danou funkci na hodnotu v každém uzlu:

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

7.2 Akumulační funkce nad vlastními datovými typy

Příklad 7.2.1 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 7.2.2 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 `07_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 7.2.3 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 []]
```

7.3 Složitější příklady

Příklad 7.3.1 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 7.3.2 Dokažte, že funkce `($) . ($) . ... . ($)` představuje pro libovolný konečný počet `($)` vždy tu stejnou funkci. Zkuste intuitivně argumentovat, proč tomu tak je.

Příklad 7.3.3 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 7.3.4 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 7.3.5 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 7.3.6 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 7.3.7 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 7.3.8 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` měly stejný typ. Zkuste najít více řešení.

Příklad 7.3.9 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 7.3.10 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 7.3.11 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 7.3.12 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 7.3.13 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 7.1.1

```
ntreeSize :: NTree a -> Integer
ntreeSize (NNode _ subtrees) = 1 + sum (map ntreeSize subtrees)

ntreeSum :: Num a => NTree a -> a
ntreeSum (NNode v subtrees) = v + sum (map ntreeSum subtrees)

ntreeMap :: (a -> b) -> NTree a -> NTree b
ntreeMap f (NNode v subtrees) = NNode (f v) (map (ntreeMap f) subtrees)
```

Řešení 7.2.1 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`. Naším 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 7.2.2

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 7.2.3

```
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 7.3.1

```
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 7.3.2 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 `dollar` $_n \equiv$ `($)`. Pak `dollar`$_{(n+1)} \equiv$ `($) . dollar` $_n \equiv$ `($) . ($)` a stačí tedy dokázat `($)` $\equiv$ `($) . ($)`. Máme

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

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

Řešení 7.3.3

```
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 7.3.4

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

Řešení 7.3.5

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]` $\rightsquigarrow$ `[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)` $\equiv$ `snd`, `uncurry const` $\equiv$ `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 7.3.6

```
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 7.3.7

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 7.3.8

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

Řešení 7.3.9 `f = \_ (x:y:s) -> (x + y) : (x:y:s)`

Řešení 7.3.10 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 7.3.11 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 7.3.12 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 7.3.13 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.