technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Alexander Becker (*alexander2.becker@tu-dortmund.de*)
Sascha Mücke (*sascha.muecke@tu-dortmund.de*)

Wintersemester 2016/2017

Freiwillige Zusatzübungen zu Funktionaler Programmierung

Ausgabe: 19. Januar 2017, **Abgabe:** keine

Aufgabe Z.1 Implizite Klammerung von Funktionen und Typen
Ergänzen Sie die impliziten Klammern folgender Funktionstypen.

```
(*) :: Num a => a -> a -> a
(.) :: (b -> c) -> (a -> b) -> a -> c
filter :: (a -> Bool) -> [a] -> [a]
add5Ints :: Int -> Int -> Int -> Int -> Int -> Int
```

Ergänzen Sie die impliziten Klammern folgender Funktionen.

```
(*) 3 4
(.) f g
filter even [1..10]
add5Ints 1 2 3 4 5
map (\x -> \y -> \z -> x:y:z:[]) . filter even
\x -> \y -> g x $ f . h y
```

Lösungsvorschlag

```
(*) :: Num a => a -> (a -> a)
(.) :: (b -> c) -> ((a -> b) -> (a -> c))
filter :: (a -> Bool) ->([a] -> [a])
add5Ints :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))

((*) 3) 4
((.) f) g
(filter even) [1..10]
((((add5Ints 1) 2) 3) 4) 5
(map (\x -> (\y -> (\z -> x:(y:(z:[]))))))) . (filter even)
\x -> (\y -> (g x) $ (f . (h y)))
```

Aufgabe Z.2 Typinferenz
Bestimmen Sie die Funktionstypen folgender Funktionen so allgemein wie möglich.

```
\x -> x + 1
filter id
sum . filter odd
map ($ 'x')
map (\x -> Wrap $ (even . length) x)
flip $ flip bexp2store x
```

Zur Bestimmung verwenden Sie:

```
bexp2store :: BExp a -> BStore a -> Store a -> Bool
data WrapBool = Wrap Bool
```

Lösungsvorschlag

```
\x -> x + 1 :: Num a => a -> a
filter id :: [Bool] -> [Bool]
sum . filter odd :: Integral a => [a] -> a
map ($ 'x') :: [Char -> a] -> [a]
map (\x -> Wrap $ (even . length) x) :: [[a]] -> [WrapBool]
flip $ flip bexp2store x :: Store a -> BExp a -> Bool
```

Aufgabe Z.3 Listen

- Schreiben sie eine Funktion, die die unendliche Liste aller geraden Fibonacci-Zahlen zurückgibt.
- Schreiben Sie eine Funktion, die überprüft, ob eine gegebene Liste ein Palindrom ist.
- Schreiben Sie eine Funktion, die alle Elemente der Collatzfolge enthält, beginnend bei einem Startwert x.
- Implementieren Sie den Quicksort-Algorithmus auf Listen.
  Hinweis: Nutzen Sie als Pivotelement das erste Element der Liste.
- Schreiben Sie eine Funktion, die die Quadrate aller Zahlen von 1 bis x enhält.
- Schreiben Sie eine Funktion, die die unendliche Liste der pythagoreischen Tripel erzeugt. Nutzen sie dazu Listenkomprehension.
  Hinweis: Ein pythagoreisches Tripel (x, y, z) ist ein Tripel für das $x^2 + y^2 = z^2$ gilt.
- Schreiben Sie eine Funktion, die aus einer eingegeben Int-Liste alle Vielfachen von 7 entfernt. Nutzen Sie dafür die Funktion ***filter***.
- Schreiben Sie die Funktion ***updateEntries*** unter Verwendung von ***foldl***, die die Werte einer Int-Liste addiert mit den Werten der Int-Listen, die jeweils an selber Stelle stehen.
  Beispiel:

  ```
  updateEntries :: [Int] -> [[Int]] -> [Int]
  updateEntries [1,4,2] [[1,0,3], [0, -2, 1]] = [2,2,6]
  ```

- Schreiben Sie die Funktion ***compare3***, die drei Listen als Eingabe erwartet. Verglichen werden dann jeweils die Elemente der Listen, die an selber Stelle stehen. Gilt x <= y <= z, soll die Ergebnisliste an selber Stelle True enhalten, ansonsten False.
  Beispiel:

  ```
  compare3 :: Eq a => [a] -> [a] -> [a] -> [Bool]
  compare3 [1,2,3] [2,3,4] [5,1,5] = [True, False, True]
  ```

Lösungsvorschlag

```
evenFibs :: [Integer]
evenFibs = filter even fibs

fibs :: [Integer]
fibs = 1:2:f 1 2 where
f x y = let z = x+y in z:f y z

isPalindrom :: Eq a => [a] -> Bool
isPalindrom list = and $ zipWith (==) list $ reverse list
```

```
isPalindrom :: Eq a => [a] -> Bool
isPalindrom list = list == reverse list

collatz :: Int -> [Int]
collatz n
        | even n = n : collatz (div n 2)
        | otherwise = n : collatz (3*n+1)

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = let
        leq = [ x | x <- xs, x <= p]
        g = [ x | x <- xs, x > p] in
                quicksort leq ++ [p] ++ quicksort g

squares :: Int -> [Int]
squares n = [ x^2 | x <- [0..n]]

pythagoreanTriplets :: [(Int,Int,Int)]
pythagoreanTriplets = [(x,y,z) | z <- [1..] , y <- [1..z] , x <- [1..y] , x

multiplesOfSeven :: [Int] -> [Int]
multiplesOfSeven xs = filter (\x -> not $ mod x 7 == 0) xs

updateEntries :: [Int] -> [[Int]] -> [Int]
updateEntries ws xs = foldl (zipWith (+)) ws xs

compare3 :: Ord a => [a] -> [a] -> [a] -> [Bool]
compare3 xs ys zs = zipWith3 (\x y z -> x <= y && y <= z) xs ys zs
```

Aufgabe Z.4 Datentypen und Typklassen

- Schreiben Sie einen Datentyp ***List a*** analog zum Haskell-Standardtyp für Listen.
- Definieren Sie eine ***Eq***-Instanz für ***List a***.
- Definieren Sie eine ***Ord***-Instanz für ***List a***, sodass folgendes Verhalten erreicht wird:

  ```
  "hallo" <= "hello" = True
  [1,2,3,4,5] <= [1,2,3,5,5] = True
  [1,3,4,5,6] <= [1,2,3,4,5] = False
  ```

- Definieren Sie eine ***Show***-Instanz für ***List a*** analog zu dem standard Listendatentyp.

Gegeben seien folgende Datentypen:

```
data PosNat = One | Succ' PosNat
data Int' = Zero' | Plus PosNat | Minus PosNat
data Q = Div Int' PosNat
```

- Definieren Sie eine ***Eq***-Instanz für ***Q***.
- Definieren Sie eine ***Ord***-Instanz für ***Q***.
- Definieren Sie eine ***Show***-Instanz für ***Q***.

Lösungsvorschlag

```
data List a = Nil | Cons a (List a)

instance Eq a => Eq (List a) where
        Nil == Nil = True
        Cons x xs == Cons y ys = x == y && xs == ys
        _ == _ = False

instance Ord a => Ord (List a) where
        Nil <= _ = True
        Cons x xs <= Cons y ys
                | x < y     = True
                | x > y     = False
                | otherwise = xs <= ys
        _ <= _ = False

instance Show a => Show (List a) where
        show list = '[' : show' list ++ "]" where
                show' :: Show a => List a -> String
                show' Nil = ""
                show' (Cons x Nil) = show x
                show' (Cons x xs) = show x ++ "," ++ show' xs


toNum :: Num a => Int' -> a
toNum Zero' = 0
toNum (Plus n) = posNatToNum n
toNum (Minus n) = - posNatToNum n

posNatToNum :: Num a => PosNat -> a
posNatToNum One = 1
posNatToNum (Succ' n) = 1 + posNatToNum n

div' :: Int' -> PosNat -> Float
div' n d = toNum n / posNatToNum d

instance Eq Q where
        Div n d == Div n' d' = div' n d == div' n' d'

instance Ord Q where
        Div n d <= Div n' d' = div' n d <= div' n' d'

instance Show Q where
        show (Div n d) = show (toNum n) ++ "␣/␣" ++ show (posNatToNum d)
```

Aufgabe Z.5 Typen von Ausdrücken
Gehen Sie für die gegebenen Ausdrücke folgendermaßen vor:

- Ergänzen Sie die impliziten Klammern!
- Geben Sie an, ob der Ausdruck auswertbar ist oder ob ein Typfehler vorliegt!
- Ist der Ausdruck auswertbar, geben Sie seinen gesamten Typ und die Typen der verwendeten Variablen (`f, x, y, z, v`) an!

- Ist der Ausdruck nicht auswertbar, begründen Sie kurz, wieso!

```
1. \f -> f + 3
2. \x -> \y -> x y 2
3. zipWith and
4. foldr (:) [0]
5. foldl (:)
6. \u v -> u $ 2 * v - 3 * u
7. flip replicate () 20
8. \f -> \x -> f x && x 5
9. concat $ map (\x -> [[x]]) [1..100]
10. filter (<= 10) "Hello world!"
```

Lösungsvorschlag

```
\f -> (f + 3) :: Num a => a -> a
f :: Num a => a

\x -> (\y -> ((x y) 2)) :: Num b => (a -> b -> c) -> a -> c
x :: Num b => a -> b -> c
y :: a

zipWith and
nicht auswertbar, da and nicht auf den von zipWith
geforderten Typ a -> b -> c passt.

(foldr (:)) [0] :: (Num a, Foldable t) => t a -> [a]

foldl (:)
nicht auswertbar, da (:) :: a -> [a] -> [a] nicht auf den von
foldl geforderten Typ b -> a -> b passt.

\u v -> (u $ ((2 * v) - (3 * u)))
nicht auswertbar, da u sich selbst als Funktionswert erhaelt
und somit eine Funktion unendlich hoher Ordnung sein muesste.

((flip replicate) ()) 20 :: [()]

\f -> (\x -> ((f x) && (x 5))) :: Num a => ((a -> Bool) -> Bool)
                                        -> (a -> Bool) -> Bool
f :: Num a => (a -> Bool) -> Bool
x :: Num a => a -> Bool

concat $ ((map (\x -> [[x]])) [1..100]) :: (Enum a, Num a) => [[a]]
x :: (Enum a, Num a) => a

(filter (<= 10)) "hello world"
nicht auswertbar, da Char nicht mit Num a => a vergleichbar ist.
```

Aufgabe Z.6 (Lazy) Auswertung von Ausdrücken
Werten Sie die folgenden Ausdrücke schrittweise aus. Wenn nötig, nutzen Sie das Prinzip der *Lazy Evaluation*! Die Funktion **`col`** sei definiert als

```
col :: Int -> Int
col = if even n then n `div` 2 else 3*n + 1
```

```
1. (\u -> u 'E') (negate . fromEnum)
2. reverse $ take 3 $ iterate pred 900
3. takeWhile (< 20) $ iterate col 18
4. (\y -> y (+)) $ (\g p -> p 3 $ g 2 2) (*)
5. foldr (++) [] [[1,2],[3,4],[5,6]]
```

Lösungsvorschlag

```
   (\u -> u 'E') (negate . fromEnum)
~> (negate . fromEnum) 'E'
~> negate (fromEnum 'E')
~> negate 69
~> -69

   reverse $ take 3 $ iterate pred 900
~> reverse $ take 3 $ 900 : iterate pred (pred 900)
~> reverse $ 900 : (take 2 $ iterate pred (pred 900)
~> reverse (take 2 $ iterate pred (pred 900)) ++ [900]
~> reverse (take 2 $ pred 900 : iterate pred (pred (pred 900))) ++ [900]
~> reverse (900 : (take 1 $ iterate pred (pred (pred 900))) ++ [900]
~> reverse (take 1 $ iterate pred (pred (pred 900))) ++ [pred 900] ++ [900]
~> reverse (take 1 $ pred (pred 900) : iterate pred (pred (pred
          (pred 900)))) ++ [pred 900] ++ [900]
~> reverse (pred (pred 900) : (take 0 $ iterate pred (pred (pred
      (pred 900))))) ++ [pred 900] ++ [900]
~> reverse (take 0 $ iterate pred (pred (pred (pred 900)))) ++
      [pred (pred 900)] ++ [pred 900] ++ [900]
~> reverse [] ++ [pred (pred 900)] ++ [pred 900] ++ [900]
~> [] ++ [pred (pred 900)] ++ [pred 900] ++ [900]
~> [pred (pred 900)] ++ [pred 900] ++ [900]
~> pred (pred 900) : ([] ++ [pred 900]) ++ [900]
~> pred (pred 900) : [pred 900] ++ [900]
~> pred (pred 900) : pred 900 : ([] ++ [900])
~> pred (pred 900) : pred 900 : [900]
~> [pred (pred 900), pred 900, 900]
~> [pred 899, 899, 900]
~> [898, 899, 900]

   takeWhile (< 20) $ iterate col 18
~> takeWhile (< 20) $ 18 : (iterate col (col 18))
~> 18 : (takeWhile (< 20) $ iterate col (col 18))
~> 18 : (takeWhile (< 20) $ col 18 : iterate col (col (col 18)))
~> 18 : (takeWhile (< 20) $ 9 : iterate col (col (col 18)))
~> 18 : 9 : (takeWhile (< 20) $ iterate col (col (col 18)))
~> 18 : 9 : (takeWhile (< 20) $ col (col 18) : iterate col (col (col
      (col 18))))
~> 18 : 9 : (takeWhile (< 20) $ col 9 : iterate col (col (col (col 18))))
~> 18 : 9 : (takeWhile (< 20) $ 28 : iterate col (col (col (col 18))))
~> 18 : 9 : []
~> [18, 9]

   (\y -> y (+)) $ (\g p -> p 3 $ g 2 2) (*)
~> (\y -> y (+)) $ \p -> p 3 $ 2 * 2)
```

```
~> (\p -> p 3 $ 2 * 2) (+)
~> (+) 3 $ 2 * 2
~> (+) 3 4
~> 7

   foldr (++) [] [[1,2],[3,4],[5,6]]
~> (++) [1,2] $ foldr (++) [] [[3,4],[5,6]]
~> (++) [1,2] $ (++) [3,4] $ foldr (++) [] [[5,6]]
~> (++) [1,2] $ (++) [3,4] $ (++) [5,6] $ foldr (++) [] []
~> (++) [1,2] $ (++) [3,4] $ (++) [5,6] []
~> (++) [1,2] $ (++) [3,4] $ 5 : ([6] ++ [])
~> (++) [1,2] $ (++) [3,4] $ 5 : 6 : ([] ++ [])
~> (++) [1,2] $ (++) [3,4] $ 5 : 6 : []
~> (++) [1,2] $ (++) [3,4] [5,6]
~> (++) [1,2] $ 3 : ([4] ++ [5,6])
~> (++) [1,2] $ 3 : 4 : ([] ++ [5,6])
~> (++) [1,2] $ 3 : 4 : [5,6]
~> (++) [1,2] [3,4,5,6]
~> 1 : ([2] ++ [3,4,5,6])
~> 1 : 2 : ([] ++ [3,4,5,6])
~> 1 : 2 : [3,4,5,6]
~> [1,2,3,4,5,6]
```

Aufgabe Z.7 Multimenge

Gegeben sei folgende Datenstruktur **`Multiset a`**, die eine Multimenge modelliert. Die Liste `items` ist die Menge der enthaltenen Werte, die Funktion `count` gibt an, wie oft ein Element in der Multimenge enthalten ist (z.B. `count ms 'G'` $\rightsquigarrow$ `4`, also `'G'` ist viermal in der Multimenge `ms` enthalten).

```
data Multiset a = Multiset {
        count :: a -> Int,
        items :: [a]
}
```

Folgende Bedingungen sollen für **`Multiset a`** immer gelten:

- `count ms x >= 0` für alle `x :: a` und `ms :: Multiset a`
- Wenn `count ms x == 0`, dann gilt `not $ x ‘elem‘ items ms`
- Wenn `count ms x > 0`, dann gilt `x ‘elem‘ items ms`
- `items ms` enthält jedes Element nur einmal

Implementieren Sie folgende Funktionen unter Beachtung der o.g. Invarianten!

1. **`empty :: Multiset a`** - eine leere Multimenge
2. **`size :: Multiset a -> Int`** - gibt die Anzahl der Elemente einer Multimenge zurück, insbesondere werden mehrfach auftretende Elemente auch mehrfach gezählt
3. **`add, remove :: Eq a => Int -> a -> Multiset a -> Multiset a`** - der Aufruf `add n x` fügt n-mal das Element x zu einer Multimenge hinzu bzw. entfernt n davon
4. **`fromList :: Eq a => [a] -> Multiset a`** - überführt eine Liste sinnvoll in eine Multimenge
5. **`fromMultiset :: Multiset a -> [a]`** - überführt eine Multimenge sinnvoll in eine Liste

6. **`unionMS, meetMS, diffMS :: Eq a => Multiset a -> Multiset a -> Multiset a`** - bildet die Vereinigung (bzw. den Schnitt, die Differenz) zweier Multimengen analog zur Mengenoperation (bzw. `meet`, `diff`)

7. **`mapMS :: Eq b => (a -> b) -> Multiset a -> Multiset b`** - analog zu `map` auf Listen. Hinweis: Die Funktion f bei einem Aufruf `mapMS f` ist i.A. *nicht* injektiv!

Lösungsvorschlag

```
-- Aendere Wert einer Funktion an einer Stelle
update :: Eq a => a -> b -> (a -> b) -> a -> b
update a b f x
  | x == a    = b
  | otherwise = f x

empty :: Multiset a
empty = Multiset (const 0) []

size :: Multiset a -> Int
size (Multiset c xs) = foldr (\x total -> total + c x) 0 xs

add, remove :: Eq a => Int -> a -> Multiset a -> Multiset a
add n x (Multiset c xs)
  | x `elem` xs = Multiset (update x (c x + n) c) xs
  | otherwise   = Multiset (update x n c) $ x : xs

remove n x ms@(Multiset c xs)
  | not $ x `elem` xs = ms
  | c x <= n          = Multiset (update x 0 c) $ filter (/= x) xs
  | otherwise         = Multiset (update x (c x - n) c) xs

fromList :: Eq a => [a] -> Multiset a
fromList = foldr (add 1) empty

fromMultiset :: Multiset a -> [a]
fromMultiset (Multiset c xs) = foldr (\x l -> replicate (c x) x ++ l) [] xs

unionMS, meetsMS, diffMS :: Eq a => Multiset a -> Multiset a -> Multiset a
unionMS (Multiset c1 xs1) (Multiset c2 xs2) = Multiset {
  count = \x -> c1 x + c2 x,
  items = xs1 `union` xs2
}

meetsMS (Multiset c1 xs1) (Multiset c2 xs2) = Multiset {
  count = \x -> min (c1 x) (c2 x),
  items = xs1 `intersect` xs2
}

diffMS (Multiset c1 xs1) (Multiset c2 xs2) = Multiset {
  count = \x -> max 0 $ c1 x - c2 x,
  items = filter (\x -> c1 x > c2 x) xs1
}

mapMS :: Eq b => (a -> b) -> Multiset a -> Multiset b
mapMS f (Multiset c xs) = foldr (\x ms -> add (c x) (f x) ms) empty xs
```

Aufgabe Z.8 Instanziierung von Typklassen
Gegeben sei die folgende Typklasse **Stack**, die typische Operationen eines Kellerspeichers bereitstellt.

```
class Stack s where
  sempty :: s a
  push   :: a -> s a -> s a
  pop    :: s a -> Maybe (a, s a)
  peek   :: s a -> Maybe a
```

1. Schreiben Sie eine Stack-Instanz für `[]`.
2. Schreiben Sie eine Stack-Instanz für **Colist**.
   Zur Erinnerung: `Colist a = Colist { split :: Maybe (a, Colist a) }`
3. Schreiben Sie eine Funktion **`fromStack :: (Eq a, Stack s) => s a -> Multiset a`**, die alle Elemente eines Stacks in eine Multimenge legt.
4. Schreiben Sie eine Funktion **`reverseStack :: Stack s => s a -> s a`**, die die Reihenfolge der Elemente eines Stacks umkehrt.
5. Schreiben Sie eine Funktion **`pops :: Stack s => Int -> s a -> ([a], s a)`**, sodass `pops n` die ersten *n* Elemente von einem Stack entfernt und in einer Liste zusammen mit dem Reststack zurückgibt. Ist *n* größer als die Anzahl *m* der Elemente auf dem Stack, sollen die *m* Elemente zusammen mit dem leeren Stack zurückgegeben werden, insbesondere soll bei einem leeren Stack die Rückgabeliste leer sein.

Lösungsvorschlag

```
instance Stack [] where
  sempty      = []
  push        = (:)
  pop (x:xs)  = Just (x, xs)
  pop _       = Nothing
  peek (x:xs) = Just x
  peek _      = Nothing

instance Stack Colist where
  sempty    = Colist Nothing
  push x cl = Colist $ Just (x, cl)
  pop       = split
  peek (Colist (Just (x, cl))) = Just x
  peek _                       = Nothing

fromStack :: (Eq a, Stack s) => s a -> Multiset a
fromStack = fromStack' empty where
  fromStack' ms s = case pop s of
    Just (x, s') -> fromStack' (add 1 x ms) s'
    _            -> ms

reverseStack :: Stack s => s a -> s a
reverseStack = revStack sempty where
  revStack sTo sFrom = case pop sFrom of
    Just (x, s') -> revStack (push x sTo) s'
    _            -> sTo
```

```
pops :: Stack s => Int -> s a -> ([a], s a)
pops n s = pops' n ([], s) where
  pops' 0 (l, s) = (l, s)
  pops' n (l, s) | n > 0 = case pop s of
    Just (x, s') -> pops' (n-1) (x : l, s')
    _            -> (l, s)
```