04IN1023: Grundlagen der funktionalen Programmierung

Klausur SoSe 2022

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel,
M.Sc., M.Ed. Hakan Aksu
29 Juli 2022

Name, Vorname	
Matrikel-Nr.	
Email	.....@uni-koblenz.de
Studiengang	☐ BSc Inf ☐ BSc CV ☐ .....
Prüfungsversuch	☐ 1 ☐ 2 ☐ 3 ☐ 4
Prüfungsordnung	☐ >= $PO2019$ ☐ < $PO2019$

Hiermit bestätige ich, dass ich zur Klausur angemeldet und zugelassen bin.
Eine falsche Angabe wird als Täuschungsversuch gewertet.

Unterschrift: _____

Korrekturabschnitt

Aufgabe	**Punkte (0-2)**
1	
2	
3	
4	
5	
6	
7	
8	
9	

1 “Simple Algorithmen”

Gegeben ist eine Repräsentation von natürlichen Zahlen wie folgt:

```
-- Natural numbers are even or odd natural numbers
data Nat = Even Even | Odd Odd
-- Even natural numbers: 0 or successors of odd natural numbers
data Even = Zero | SuccO2E Odd
-- Odd natural numbers: 1 or successors of even natural numbers
data Odd = One | SuccE2O Even
```

Gesucht ist eine Funktion `nat2int` zur Konvertierung der Repräsentation nach `Int` – illustriert wie folgt:

```
nat2int :: Nat -> Int

testsNat = TestList [
    0 ~=? nat2int (Even Zero),
    1 ~=? nat2int (Odd One),
    3 ~=? nat2int (Odd (SuccE2O (SuccO2E One)))
 ]
```

Reference solution

```
nat2int :: Nat -> Int
nat2int n = case n of
   (Even e) -> even2int e
   (Odd o) -> odd2int o
  where
   even2int :: Even -> Int
   even2int Zero = 0
   even2int (SuccO2E o) = odd2int o + 1
   odd2int :: Odd -> Int
   odd2int One = 1
   odd2int (SuccE2O e) = even2int e + 1
```

2 “Suchen und Sortieren”

Gegeben ist eine mögliche Implementation von Selection Sort:

```
selection_sort [] = []
selection_sort (x:xs) = min : selection_sort rest
  where
    (min, rest) = split x xs
```

Hier wird eine Funktion `split` vorausgesetzt, welche das Minimum in einer gegebenen Liste ermittelt und auch die verbleibenden Elemente zurückgibt. (Die Funktion nimmt auch immer ein bisheriges Minimum als Argument, um auch für Listen der Länge 0 total definiert zu sein.)

Gesucht werden zwei Funktionen:

- eine Implementation von `split` sowie
- eine Variation von Selection Sort, `selection_sort_pfx`, welche auch gern unter Verwendung von `split`, aber ohne Verwendung der Originalfunktion `selection_sort` nur den Prefix der Ausgabe von einer gewissen Länge ermittelt.

Dies wird illustriert wie folgt:

```
testsSorting = TestList [
    (1,[8,3,10,5]) ~=? split 8 [3,1,10,5],
    [1,3,5,8,10] ~=? selection_sort [8,3,1,10,5],
    [1,3,5] ~=? selection_sort_pfx 3 [8,3,1,10,5]
  ]
```

Reference solution

```
split x [] = (x, [])
split x (x':xs)
  = if x <= x'
       then let (m, r) = split x xs in (m, x':r)
       else let (m, r) = split x' xs in (m, x:r)

selection_sort_pfx 0 xs = []
selection_sort_pfx n [] = []
selection_sort_pfx n (x:xs) = min : selection_sort_pfx (n−1) rest
  where
    (min, rest) = split x xs
```

3 “Simple Datenmodelle”

Gesucht sind

- Rekordtypen `C1`, `C2` und `C3` für 1-, 2- und 3-dimensionale Koordinaten, wobei die Koordinaten (x, y, z) Floats sein sollen;
- ein Datentyp `C` welcher `C1`, `C2` und `C3` als Varianten zusammenfasst;
- eine Funktion `c1_c2` welche eine 1-dimensionale Koordinate und einen Wert für die zweite Dimension in eine 2-dimensionale Koordinate umwandelt – siehe nachfolgende Signatur.

```
c1_c2 :: C1 -> Float -> C2
```

Reference solution

```
data C1 = C1 { c1_x :: Float }
data C2 = C2 { c2_x :: Float, c2_y :: Float }
data C3 = C3 { c3_x :: Float, c3_y :: Float, c3_z :: Float }
data C = C1' C1 | C2' C2 | C3' C3

c1_c2 :: C1 -> Float -> C2
c1_c2 c f = C2 (c1_x c) f
```

4 “Abstrakte Datentypen”

Gesucht wird ein ADT `Counter` mit Operationen `getInt`, `zero`, `inc` (“increment”) und `dec` (“decrement”), wobei die Operation `dec` keine negativen Werte zulässt – illustriert wie folgt:

```
zero :: Counter
inc :: Counter -> Counter
dec :: Counter -> Counter
getInt :: Counter -> Int

testsCounter = TestList [
    0 ~=? getInt zero,
    2 ~=? getInt ((inc . inc) zero),
    0 ~=? getInt ((dec . dec) zero)
 ]
```

Bitte Folgendes beachten! Die Lösung muß als Modul für den ADT angegeben werden. Dabei muß der ADT dafür sorgen, dass von außen nur über die genannten Operation auf Werte vom Typ `Counter` zugegriffen werden kann.

Reference solution

```
module Counter(
   Counter,
   getInt,
   zero,
   inc,
   dec
) where

newtype Counter = Counter { getInt :: Int }

zero :: Counter
zero = Counter 0

inc :: Counter -> Counter
inc (Counter x) = Counter (x+1)

dec :: Counter -> Counter
dec (Counter x) | x == 0 = Counter 0
dec (Counter x) | x > 0 = Counter (x-1)
```

5 “Funktionale Datenstrukturen”

Gegeben ist ein Datentyp für unendliche Binärbäume:

```
data BinTree = BinTree Int BinTree BinTree
```

Zur Illustration folgt hier auch Code zur Konstruktion eines Beispielbaums aus einer unendlichen Liste:

```
sample_tree :: BinTree
sample_tree = list2tree [0..]

list2tree :: [Int] -> BinTree
list2tree xs = BinTree x (list2tree ys) (list2tree zs)
 where
  (x, ys, zs) = fork xs

fork :: [Int] -> (Int, [Int], [Int])
fork (x:xs) = (x, fst v, snd v)
 where
  v = fork' xs
  fork' (x:x':xs) = (x:fst w, x':snd w) where w = fork' xs
```

Gesucht ist nun die Funktion `join` innerhalb der folgenden Funktion `tree2list`, welche eine Binärbaum in eine Liste umwandelt – illustriert wie folgt:

```
tree2list :: BinTree -> [Int]
tree2list (BinTree x l r) = join x (tree2list l) (tree2list r)

join :: Int -> [Int] -> [Int] -> [Int]

testsTree = TestList [
   [0,1,2,3,4] ~=? take 5 (tree2list sample_tree)
 ]
```

Zu beachten ist hier, dass `join` nicht einfach die Teillisten verketten kann, da sonst der unendliche linke Teilbaum es unmöglich macht, den rechten Teilbaum überhaupt jemals in der resultierenden Liste anzuschauen.

Reference solution

```
join :: Int -> [Int] -> [Int] -> [Int]
join x ys zs = x:join' ys zs
 where
  join' (y:ys) (z:zs) = y:z:join' ys zs
```

6 “Funktionen höherer Ordnung”

Gesucht wird eine Funktion `sequ`, die eine gegebene Liste von Funktionen der Reihe nach von links nach rechts anwendet – illustriert wie folgt:

```
sequ :: [(a -> a)] -> a -> a

testsSequ = TestList [
    42 ~=? sequ [] 42,
    42 ~=? sequ [(+1), (+2), (+3)] 36,
    1 ~=? sequ [const 0, (+1)] 42
 ]
```

Zu beachten ist, dass `sequ` nicht rekursiv durch Behandlung der Muster von Listen zu definieren ist, sondern durch Verwendung von `foldl` oder `foldr`, deren Typen hier in Erinnerung gerufen werden:

```
> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Reference solution

```
sequ :: [(a -> a)] -> a -> a
sequ = foldr (flip (.)) id
```

7 “Typ-Klassen Polymorphismus”

Gegeben ist eine Typklasse wie folgt:

```
class Size a
 where size :: a -> Int
```

Gesucht sind Instanzen für die im Folgenden exerzierten Testfälle:

```
testsSize = TestList [
    1 ~=? size (42::Int),
    4 ~=? size ([(1, 2), (3, 4)]::[(Int, Int)])
 ]
```

Reference solution

```
instance Size Int
 where size = const 1
instance Size a => Size [a]
 where size = sum . map size
instance (Size a, Size b) => Size (a, b)
 where size (x, y) = size x + size y
```

8 “Functors & Friends”

Gegeben ist ein Datentyp für nichtleere Listen mit einem Cursor auf einem Element:

```
data Cursor a = Cursor {
  getBefore :: [a],
  getCursor :: a,
  getAfter :: [a]
} deriving (Eq, Show)
```

Gesucht werden `Functor`- und `Foldable`-Instanzen für `Cursor` – illustriert wie folgt:

```
testsCursor = TestList [
    Cursor [] (2::Int) [3] ~=? fmap (+1) (Cursor [] (1::Int) [2]),
    3 ~=? foldr (+) 0 (Cursor [] (1::Int) [2])
 ]
```

Reference solution

```
instance Functor Cursor
 where
  fmap f x = Cursor y1 y2 y3
   where
    y1 = fmap f $ getBefore x
    y2 = f $ getCursor x
    y3 = fmap f $ getAfter x

instance Foldable Cursor
 where
  foldr f z x = foldr f z (x1++x2:x3)
   where
    x1 = getBefore x
    x2 = getCursor x
    x3 = getAfter x
```

9 “Unparsing & Parsing”

Gegeben ist eine der Illustration dienende Funktion `parseInt`, welche eine Sequenz aus Ziffern erwartet und ein `Int` zurückgibt.

```
parseInt :: Parsec String () Int
parseInt
  = many1 digit >>= \ds ->
    return (read ds :: Int)
```

Gesucht ist eine ähnliche Funktion `parseFloat`, welche zusätzlich einen Punkt (“.”) zur Abtrennung der Nachkommastellen und dann eben eine nichtleere Sequenz aus Nachkommastellen erwartet, um schließlich ein `Float` zurückzugeben – illustriert wie folgt:

```
parseFloat :: Parsec String () Float

testsParsing = TestList [
    Right (42::Int) ~=? runP parseInt () "" "42",
    Right (42.0::Float) ~=? runP parseFloat () "" "42.0"
  ]
```

Reference solution

```
parseFloat :: Parsec String () Float
parseFloat
  = many1 digit >>= \ds1 ->
    char '.' >>= \p ->
    many1 digit >>= \ds2 ->
    return (read (ds1++p:ds2) :: Float)
```