Klausur: Programmierung und Modellierung Bry 2016

July 29, 2016

Gedankenprotokoll

Aufgabe 1 - Multiple Choice

Geben sie den allgemeinsten Typen für die folgenden Ausdrücke an:

a) `f x y = (3 * x :: Int, y)`

- ☐ `Int -> Int -> (Int, Int)`
- ☐ `Int -> a -> (Int, a)`
- ☐ `(Int, Int) -> (Int, Int)`
- ☐ Kein korrekter Haskell Ausdruck
- ☐ Nichts davon, sondern .....

Lösung:

```haskell
-- f nimmt zwei Argumente und gibt etwas zurück
f : a -> b -> c
-- x wird multipliziert
f :: Num a => a -> b -> c
-- x wird als Int festgelegt (hat eine Num Instanz)
f :: Int -> b -> c
-- f gibt ein Tupel zurück, mit erstem Element x
f :: Int -> b -> (Int, b)
-- a für b einsetzen, da es ein beliebiger Name ist
f :: Int -> a -> (Int, a)
```

b) `(\x y -> (x + y) `mod` y)`

- ☐ `Int -> Int -> Int`
- ☐ `Integer -> Integer -> Integer`
- ☐ `Integral a => a -> a -> a`
- ☐ Kein korrekter Haskell Ausdruck
- ☐ Nichts davon, sondern .....

Lösung:

```haskell
-- man kann diese anonyme Funktion umschreiben in:
f x y = (x + y) `mod` y
-- Funktion nimmt zwei Argumente und gibt was zurück
f :: a -> b -> c
-- x und y werden addiert => gleicher Typ + Num
Num a => a -> a -> c
-- auf das Ergebnis wird 'mod' angewandt
Integral a => a -> a -> c
-- das ist auch der Rückgabewert der Funktion
Integral a => a -> a -> a
```

c) `(\x -> (\y -> x ++ y))`

- ☐ `String`
- ☐ `Char -> String`
- ☐ `[a] -> [a] -> [a]`
- ☐ Kein korrekter Haskell Ausdruck
- ☐ Nichts davon, sondern .....

Lösung:

```haskell
-- man kann diese anonyme Funktion umschreiben in:
(\x y -> x ++ y)
-- oder:
f x y = x ++ y
-- f nimmt zwei Argumente und gibt eins zurück:
f :: a -> b -> c
-- x und y werden mit ++ addiert
-- => Listenverknüpfung + gleicher Typ
f :: [a] -> [a] -> c
-- das ist auch der Rückgabewert
f :: [a] -> [a] -> [a]
```

d) `(\x -> [x])`

- ☐ `Char -> String`
- ☐ `a -> [a]`
- ☐ `a -> a`
- ☐ Kein korrekter Haskell Ausdruck
- ☐ Nichts davon, sondern .....

Lösung:

```haskell
-- man kann diese anonyme Funktion umschreiben in:
f x = [x]
-- f nimmt ein Argument und gibt eins zurück:
f :: a -> b
-- x wird in einen Listenkonstruktor gepackt und
-- ist auch der Rückgabewert
f :: a -> [a]
```

Aufgabe 2 - Rekursion

Definieren sie für $n \in \mathbb{N} \setminus \{0\}$:

$\prod_{i=1}^{i=1} i = 1$ für $n = 1$

$\prod_{i=1}^{i=n} i = n * \prod_{i=1}^{i=n-1} i$ für $n \geq 1$

a) Eine rekursive Funktion wie oben beschrieben mit der folgenden Typsignatur. Sie soll für $n \leq 0$ nicht terminieren.

```haskell
produkt :: Integral a => a -> a
```

Lösung:

```haskell
produkt :: Integral a => a -> a
produkt 1 = 1
produkt n = n * produkt (n-1)
```

b) Erweitern sie die Funktion aus *a)* sodass sie für Eingaben $n \leq 0$ eine 1 zurückgibt.

Lösung:

```haskell
produkt :: Integral a => a -> a
produkt n
    | n <= 1 = 1
produkt n = n * produkt (n-1)
```

c) Schreiben sie eine Funktion die sich wie die Funktion aus *b)* verhält, bloß endrekursiv.

Lösung:

```haskell
produkt :: Integral a => a -> a
produkt n = go n 1
    where
        go :: Integral a => a -> a -> a
        go n acc
            | n <= 1 = acc
        go n acc = go (n-1) (acc*n)

-- Hier sind viele Lösungen möglich

produkt :: Integral a => a -> a
produkt n = foldl go 1 [1..n]
    where
        go :: Integral a => a -> a -> a
        go acc n
            | n <= 1    = acc
            | otherwise = acc * n
```

Aufgabe 3 - Auswertungsreihenfolgen

Umgebung soll nicht mit angegeben werden. Gegeben:

```
tail [1,2,3] = [2,3]
f = tail . tail
g = (\x -> 42)
h = (\x -> x * x)
```

a) Werten sie `g (h 3)` in applikativer Reihenfolge aus

Lösung:

```
    g (h 3)
=>  g ((\x -> x * x) 3)
=>  g (3 * 3)
=>  g 9
=>  (\x -> 42) 9
=>  4
```

b) Werten sie `g (h 3)` in normaler Reihenfolge aus

Lösung:

```
    g (h 3)
=>  (\x -> 42) (h 3)
=>  42
```

c) Werten sie `h (h 3)` in verzögerter Reihenfolge aus

Lösung:

```
     h (h 3)
=>  let a = h 3 in h a
=> (\x -> x * x) a
=> ^a * ^a
    => a
    => h 3
    => let b = 3 in h b
    => (\x -> x * x) b
    => ^b * ^b
    => 3 * 3
    => 9
=> 9 * 9
=> 81

-- Wahrscheinlich wäre folgendes auch ''korrekt''
-- weil es nie ausführlich definiert war
     h (h 3)
=>  let a = h 3 in h a
=> (\x -> x * x) a
=> ^a * ^a
    => a
    => h 3
    => (\x -> x * x) 3
    => 3 * 3
    => 9
=> 9 * 9
=> 81
```

d) Werten sie `f [1,2,3]` in verzögerter Reihenfolge aus

Lösung:

```
-- Hier kompiliert einiges nicht, wenn ich es
-- in GHCi einsetze, aber der Einfachkeit halber
-- ist die erste Lösung pseudokorrekt

    f [1,2,3]
=>  tail . tail [1,2,3]
=>  tail [2,3]
=>  [3]

--
    f [1,2,3]
=> (\x -> tail . tail $ x) [1,2,3] -- i)
=> tail . tail $ [1,2,3]           -- i)
=> let (x:xs) = [1,2,3] in (tail . tail $ (x:xs))
=> tail xs
=> let (y:ys) = xs in tail (y:ys)
=> ys

-- i) $ weil es sonst nicht kompiliert

-- Anmerkungen:
--
-- *) Hab hier einfach vergessen dass wir weder
-- ``.'' definiert haben, noch es in Lambda-Form
-- gebracht haben
-- *) Genauso wie wir die Definition von ``tail''
-- raten weil wir die ``let''-Ausdrücke definieren,
-- sodass sie an den Pattern-Match passen
```

Aufgabe 4 - Binärbäume

Gegeben:

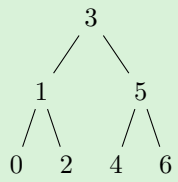
```
data BB a = L | K (BB a) a (BB a)
```

a) Geben sie einen ausgeglichenen Binärbaum vom Typ

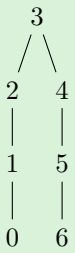
```
BB Integer
```

für die Werte $0 - 6$ (eingeschlossen) an.

Lösung:



Ausgeglichen bedeutet dass jeder Knoten zu seinen Kindknoten maximal einen Höherunterschied von 1 haben darf. Folgendes ist falsch:



Komischerweise ist die binäre Suchbaumeigenschaft nicht gefragt. Dennoch werde ich mal den ersten Baum hinschreiben:

```
K (K (K L 0 L)
      1
      (K L 2 L))
  3
  (K (K L 4 L)
      5
      (K L 6 L))
```

b) Füllen sie die Lücken aus. Gefragt ist eine rekursive Suchfunktion für Elemente in einem Baum

```
suche :: Eq a => a -> BB a -> Bool
suche .....  ......... = False
suche .....  ......... = ..........................
                         ..........................
```

Lösung:

```
suche :: Eq a => a -> BB a -> Bool
suche _   L          = False
suche e   (K l x r)  = e == x
                     || suche e l
                     || suche e r
```

Aufgabe 5 - Faltung

Gegeben:

```
data BB a = L | K (BB a) a (BB a)

tief :: b -> (b -> a -> a -> a) -> BB a -> b
tief fL fK L = fL
tief fL fK (K linkerBaum w rechterBaum) =
        fk (tief fL fK linkerBaum) w (tief fL fK rechterBaum)
```

Vervollständigen sie die Definitionen:

a) Eine Funktion die die Anzahl der Knoten in einem Binärbaum zurückgibt

```
anzahlKnoten :: BB a -> Int
anzahlKnoten baum =
    tief 0 (\left w right -> ........................... ) baum
```

Lösung:

```
anzahlKnoten :: BB a -> Int
anzahlKnoten baum =
    tief 0 (\left w right -> 1 + left + right) baum
```

b) Eine Funktion die die Tiefe eines Binärbaum zurückgibt

```
baumTiefe :: BB a -> Int
baumTiefe baum =
    tief 0 (\left w right -> ........................... ) baum
```

Lösung:

```
baumTiefe :: BB a -> Int
baumTiefe baum =
    tief 0 (\left w right -> 1 + max left right) baum
```

c) Eine Funktion die überprüft ob ein Element in einem Baum vorhanden ist. Ergänzen sie die Typsignatur.

```
istIn :: Eq a => a -> BB a -> ........................
istIn wert baum =
    tief False (\left w right -> ........................... ) baum
```

Lösung:

```
istIn :: Eq a => a -> BB a -> Bool
istIn wert baum =
    tief False (\left w right -> wert == w || left || right ) baum
```

Aufgabe 6 - Datentypen

Gegeben (alle Namen sind `Strings`):

- Im Land L_1 besteht ein Personennamen aus
  - einem oder mehreren Vornamen und
  - einem Nachnamen
- Im Land L_2 besteht ein Personennamen aus
  - einem Vornamen und
  - einem Mittelnamen und
  - einem Nachnamen

a) Definieren sie einen rekursiven Datentyp `Vornamen` für Menschen aus L_1 mit:

- `EV` *(ein Vorname)*
- `MV` *(mehrere Vornamen)*

Lösung:

```
data Vorname = MV String Vorname
             | EV String
```

b) Stellen sie die Vornamen von einem Menschen "v1", "v2" und "v3" mithilfe von diesem Datentyp dar.

Lösung:

```
names = MV ``v1'' (MV ``v2'' (EV ``v3''))
```

c) Definieren sie eine rekursive Funktion mit folgender Typsignatur die alle Vornamen zusammensetzt und zurückgibt.

```
vDruck :: Vorname -> String
```

Lösung:

```
vDruck :: Vorname -> String
vDruck (EV name)       = name
vDruck (MV name more) = name ++ vDruck more
```

d) Definieren sie einen Datentyp `Name1` mit Konstruktor `N1` für Personen aus L_1.

Lösung:

```
data Name1 = N1 Vorname String
```

e) Definieren sie eine Funktion mit folgender Typsignatur die einen Personennamen `Name1` als `String` in der Form `Vorname(n) Nachname` zurückgibt.

```
druckt1 :: Name1 -> String
```

Lösung:

```
druckt1 :: Name1 -> String
druckt1 (N1 vorname nachname) = vDruck vorname ++ ` ` : nachname
-- oder auch
druckt1 :: Name1 -> String
druckt1 (N1 vorname nachname) = vDruck vorname ++ `` '' ++ nachname
-- oder per Hand
druckt1 :: Name1 -> String
druckt1 (N1 vorname nachname) = go vorname ++ `` '' ++ nachname
    where
        go :: Vorname -> String
        go (EV name)       = name
        go (MV name more) = name ++ go more
```

f) Definieren sie einen Datentyp `Name2` mit Konstruktor `N2` für Personen aus L_2.

Lösung:

```
data Name2 = N2 String String String
-- Hier darf weder Vorname (erlaubt mehrere)
-- noch Name1 benutzt werden (benutzt Vorname)
```

g) Definieren sie eine Funktion mit folgender Typsignatur die einen Personennamen `Name2` als String in der Form `Vorname Mittelname Nachname` zurückgibt.

```
druckt2 :: Name2 -> String
```

Lösung:

```
druckt2 :: Name2 -> String
druckt2 (N2 vor mit nach) = vor ++ ` `:mit + ` `:nach
-- oder auch
druckt2 :: Name2 -> String
druckt2 (N2 vor mit nach) = vor ++ `` '' ++ mit + `` '' ++ nach
```

h) Vervollständigen sie folgende Instanzen der Typklasse `Name`. Die Funktionen sollen analog zu `druckt1` und `druckt2` für die jeweiligen Datentypen funktionieren.

```
class Name a where

    druckt :: a -> String

instance Name Name1 where

    ....................................
    ....................................
    ....................................

instance Name Name2 where

    ....................................
    ....................................
    ....................................
```

Lösung:

```
instance Name Name1 where
    druckt x = druckt1 x

instance Name Name2 where
    druckt x = druckt2 x
```

Aufgabe 7

a) Definieren sie den Datentyp `Paar a` mit Konstruktor `P` und Typvariable `a` das ein Tupel aus Haskell simuliert. Sie dürfen den eingebauten Datentyp in Haskell jedoch **nicht** benutzen.

Lösung:

```
data Paar a = Paar a a
```

b) Ergänzen sie folgende Definition von der Addition von den Datentyp `Paar a`. Bei der Addition in Tupeln werden die Elemente nach dem Index addiert (`(x1,y1) + (x2,y2) = (x1+x2, y1+y2)`)

```
plus :: .............. => Paar a -> Paar a -> Paar a
plus ............... ................ = P (x1 + x2) (y1 + y2)
```

Lösung:

```
plus :: Num a => Paar a -> Paar a -> Paar a
plus (P x1 y1) (P x2 y2) = P (x1 + x2) (y1 + y2)
```

c) Ergänzen sie die Monoidinstanz für `Pair a` sodass `a` eine Instanz in der Typklasse `Integral` besitzt und eine Monoid bezöglich der Addition ist.

```
instance ............ => Monoid (Paar a) where

    mempty = ............................
    mappend p1 p2 = ....................
```

Lösung:

```
instance Integral a => Monoid (Paar a) where

    mempty = P 0 0
    mappend p1 p2 = plus p1 p2

-- oder für mappend (geht analog auch mit let)
    mappend p1 p2 = P (x1 + x2, y1 + y2)
        where
            (P x1 y1) = p1
            (P x2 y2) = p2
```

Aufgabe 8

a) Vervollständigen sie die folgende Funktion `minus` die die Differenz von $n_1, n_2 \in \mathbb{Z} : n_1 - n_2$ nur dann berechnet, wenn $n_1 \geq 0, n_2 \geq 0, n_1 \geq n_2$. Wenn die nicht gilt soll der Fehlerwert von `Maybe Int` zurückgegeben werden. Das erste Argument ist n_1 und das zweite n_2.

```
minus :: Maybe Int -> Maybe Int -> Maybe Int
minus Nothing _ = ......................
minus _ Nothing = ......................
minus (Just n)  _ | n < 0 = ......................
minus _  (Just n) | n < 0 = ......................
minus (Just n1) (Just n2) | ......................
    = Nothing
minus (Just n1) (Just n2) | ......................
    = ..........
```

Lösung:

```
minus :: Maybe Int -> Maybe Int -> Maybe Int
minus Nothing _ = Nothing
minus _ Nothing = Nothing
minus (Just n)  _ | n < 0 = Nothing
minus _  (Just n) | n < 0 = Nothing
minus (Just n1) (Just n2) | n1 < n2
    = Nothing
minus (Just n1) (Just n2) | otherwise -- i)
    = Just (n1 - n2)

-- i) Hier kann man reinschreiben was man will,
-- solange es zu True auswertet
```

b) Ist der Datentyp `Maybe Int` mit `mappend = minus` ein Monoid? Kreuzen sie an und vervollständigen sie wenn nötig.

☐ Monoid, mit Neutrum .....

☐ ein Monoid

Lösung:

Gegenbeispiel: $(1-1)-1 \neq 1-(1-1)$ *(Assoziativität)*
Weil die Assoziativität verletzt ist, formt (Maybe Int, $-$) kein Monoid