RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

WS 2013/14
Übungsblatt 10
13.01.2014

Übung zur Vorlesung Programmierung

Aufgabe T19

Es seien `x`, `y`, `z` ganze Zahlen vom Typ `Int` und `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`.
Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[1,2,3],[4,5]]` hat den Typ `[[Int]]` und enthält 2 Elemente.

a) `[x] = x:[]`

b) `[] ++ [xs] = []:[xs]`

c) `[[]] ++ [x] = []:[x]`

Lösungsvorschlag

a) Die Notation `[x]` ist lediglich eine Kurzschreibweise für `x:[]`. Die Ausdrücke sind also gleich. Das Ergbenis ist eine einelementige Liste vom Typ `[Int]`.

b) Beide Ausdrücke haben den Typ `[[Int]]`. Jedoch hat die erste Liste ein Element, während die zweite Liste zwei Elemente besitzt. Die Ausdrücke sind also nicht gleich.

c) Beide Ausdrücke sind nicht typkorrekt. Daher würde die Gleichung in Haskell nicht gelten. Allerdings würden beide Ausdrücke die gleiche (ungültige) Liste darstellen, nämlich `[[],x]`. Diese Liste ist nicht typkorrekt, da sie sowohl eine Liste als auch einen `Int` Wert enthält.

Aufgabe T20

Gegeben sei das folgende Haskell-Programm:

```
produkt :: [Int] -> Int
produkt []     = 1
produkt (x:xs) = x * produkt xs

jedesZweite :: [Int] -> [Int]
jedesZweite []       = []
jedesZweite [x]      = [x]
jedesZweite (x:y:ys) = x : jedesZweite ys

minus10 :: [Int] -> [Int]
minus10 []     = []
minus10 (x:xs) = x - 10 : minus10 xs
```

Die Funktion `produkt` multipliziert die Elemente einer Liste miteinander, beispielsweise ergibt `produkt [3,5,2,1]` die Zahl `30`. Die Funktion `jedesZweite` bekommt eine Liste als Eingabe und gibt die gleiche Liste zurück, wobei jedes zweite Element gelöscht wurde. So ergibt `jedesZweite [1,2,3]` die Liste `[1,3]`. Die Funktion `minus10` subtrahiert von jedem Element einer Liste 10.
Geben Sie alle Zwischenschritte des Ausdrucks `produkt (jedesZweite (minus10 [3,2,1]))` bei der Auswertung an. Um Platz zu sparen, schreiben Sie hierbei `p`, `j` und `m` statt `produkt`, `jedesZweite` und `minus10`.
Hinweis: Beachten Sie, dass Haskell eine Leftmost-Outermost Auswertungsstrategie besitzt. Allerdings sind Operatoren wie `*` und `+`, die auf eingebauten Zahlen arbeiten, strikt, d. h. hier müssen vor Anwendung des Operators seine Argumente vollständig ausgewertet worden sein (wobei zunächst das linke Argument ausgewertet wird).

Lösungsvorschlag

```
p (j (m [3,2,1]))
→ p (j (3 - 10 : m [2,1]))
→ p (j (3 - 10 : 2 - 10 : m [1]))
→ p (3 - 10 : j (m [1]))
→ (3 - 10) * p (j (m [1]))
→ (-7) * p (j (m [1]))
→ (-7) * p (j (1 - 10 : m []))
→ (-7) * p (j (1 - 10 : []))
→ (-7) * p (1 - 10 : [])
→ (-7) * ((1 - 10) * p [])
→ (-7) * ((-9) * p [])
→ (-7) * ((-9) * 1)
→ (-7) * (-9)
→ 63
```

Aufgabe T21

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie für numerische Argumente den Typ `Int`. Verwenden Sie außer Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise) und Vergleichsoperatoren wie <=, ==,... **keine** vordefinierten Funktionen (dies schließt auch arithmetische Operatoren ein), außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden.

a) **`mult x y`**
Berechnet $x \cdot y$. Sie dürfen dazu + und − verwenden. Die Funktion darf sich auf negativen Eingaben beliebig verhalten.

b) **`getLastTwo xs`**
Berechnet die Teilliste der letzten zwei Elemente der `Int`-Liste `xs`. Beispielsweise berechnet `getLastTwo [12,7,23]` den Wert `[7,23]`. Die Funktion darf sich auf Listen der Länge 0 und 1 beliebig verhalten.

c) **`cleanEmpty xs`**
Die Eingabe ist eine Liste von Listen von Zahlen (vom Typ `[[Int]]`). Die Funktion berechnet die Liste von Listen von Zahlen, die aus der Eingabe entsteht, indem

alle leeren Listen aus dieser entfernt werden. Beispielsweise berechnet `cleanEmpty [[5,6],[],[8],[]]` die Liste `[[5,6],[8]]`.

d) **`nachHinten x xs`**
Verschiebt die `x` ersten Elemente der Liste `xs` an das Ende. Beispielsweise berechnet `nachHinten 2 [1,2,3,4]` die Liste `[3,4,1,2]`. Sie dürfen hier ++ und – verwenden. Die Funktion darf sich auf negativen Eingaben für `x` oder bei einer leeren Eingabeliste beliebig verhalten.

e) **`listAdd x xs`**
Addiert jeweils das n-te Listenelement auf das $(n+1)$-te Listenelement, wobei auf das erste Listenelement `x` addiert wird. Beispielsweise berechnet `listAdd 5 [1,9,3]` die Liste `[6,10,12]`. Sie dürfen hier + verwenden.

Lösungsvorschlag

```
-- a
mult :: Int -> Int -> Int
mult x 0 = 0
mult x y = x + mult x (y-1)

-- b
getLastTwo :: [Int] -> [Int]
getLastTwo [x,y]    = [x, y]
getLastTwo (x:y:ys) = getLastTwo (y:ys)

-- c
cleanEmpty :: [[Int]] -> [[Int]]
cleanEmpty []      = []
cleanEmpty ([]:xs) = cleanEmpty xs
cleanEmpty (x:xs)  = x : cleanEmpty xs

-- d
nachHinten :: Int -> [Int] -> [Int]
nachHinten n [] = []
nachHinten n xs = nachHintenH [] n xs

nachHintenH :: [Int] -> Int -> [Int] -> [Int]
nachHintenH xs 0 ys     = ys ++ xs
nachHintenH xs n (y:ys) = nachHintenH (xs ++ [y]) (n-1) ys

-- alternative Loesung:
nachHinten' :: Int -> [Int] -> [Int]
nachHinten' n []     = []
nachHinten' 0 xs     = xs
nachHinten' n (x:xs) = nachHinten' (n-1) (xs++[x])

-- e
listAdd :: Int -> [Int] -> [Int]
listAdd z []     = []
listAdd z (x:xs) = (z+x) : listAdd x xs
```

Aufgabe H25 (3+3+3 Punkte)
Es seien `x`, `y`, `z` ganze Zahlen vom Typ `Int` und `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`.
Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.

a) `[x] ++ [y] = x:[y]`

b) `[xs ++ [y]] ++ [x:ys] = [xs ++ y:[x]] ++ [ys]`

c) `x:y:z:(xs ++ ys) = [x,y,z] ++ xs ++ ys`

*Hinweis: Falls linke und rechte Seite gleich sind, genügt wiederum **eine** Angabe des Typs und der Elementzahl.*

Lösungsvorschlag

a) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die Liste `[x,y]` vom Typ `[Int]`, welche zwei Elemente enthält.

b) Die Gleichung gilt nicht. Beide Listen sind vom Typ `[[Int]]` und beide Listen haben zwei Elemente. Das erste Element der ersten Liste enthält alle Elemente aus `xs` gefolgt von `y`. Das erste Element der zweiten Liste hingegen enthält alle Element aus `xs` gefolgt von `y` und `x`. Das zweite Element der ersten Liste ist `x:ys`, wohingegen das zweite Element der zweiten Liste `ys` ist.

c) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die gleiche Liste, welche zuerst die Elemente `x`, `y`, `z`, anschließend die n Elemente der Liste `xs` und schließlich die m Elemente der Liste `ys` enthält. Diese Liste enthält also insgesamt $3 + n + m$ Elemente und ist vom Typ `[Int]`.

Aufgabe H26 (10 Punkte)
Gegeben seien folgende Funktionen.

```
progression :: [Int] -> [Int]
progression (x:xs) = x : progression ((x+1):xs)

addPrev :: Int -> [Int] -> [Int]
addPrev x (y:ys) = (x+y) : addPrev y ys

pos :: Int -> [Int] -> Int
pos 0 (x:xs) = x
pos x (y:ys) = pos (x-1) ys
```

Geben Sie alle Zwischenschritte bei der Auswertung folgendes Ausdruckes an:

```
pos 1 (addPrev 0 (progression (1:[])))
```

Lösungsvorschlag

```
pos 1 (addPrev 0 (progression (1:[])))
→ pos 1 (addPrev 0 (1 : progression ((1+1):[])))
→ pos 1 ((0 + 1) : addPrev 1 (progression ((1+1):[])))
→ pos (1-1) (addPrev 1 (progression ((1+1):[])))
→ pos 0 (addPrev 1 (progression ((1+1):[])))
→ pos 0 (addPrev 1 ((1+1) : progression (((1+1)+1):[])))
→ pos 0 ((1+(1+1)) : (addPrev (1+1) (progression (((1+1)+1):[]))))
→ (1+(1+1))
→ (1+2)
→ 3
```

Aufgabe H27 (10 Punkte)[1]

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie für numerische Argumente den Typ `Int`. Verwenden Sie außer Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise) und Vergleichsoperatoren wie <=, ==,... **keine** vordefinierten Funktionen (dies schließt auch arithmetische Operatoren ein), außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden.

a) **`millimeter f z`**
Gibt (halbwegs präzise) aus, wie viele Millimeter `f` Fuß und `z` Zoll sind. Gehen Sie hierbei davon aus, dass ein Fuß 305mm entspricht und ein Zoll genau 25mm ist. So berechnet zum Beispiel `millimeter 5 4` den Wert `1625`. Die Funktion darf sich auf negativen Eingaben beliebig verhalten. Sie dürfen hier `+` und `*` verwenden.

b) **`singletons x`**
Berechnet eine Liste mit x Elementen, wobei jedes Listenelement eine einelementige Liste ist. Die Elemente der einelementigen Listen sind die Zahlen x, $x-1$, ..., 1. Beispielsweise berechnet `singletons 3` die Liste `[[3],[2],[1]]`. Die Funktion darf sich auf negativen Eingaben beliebig verhalten. Sie dürfen hier `-` verwenden.

c) **`kleinstes xs`**
Berechnet das kleinste Element, das in der `Int`-Liste `xs` vorkommt. Zum Beispiel liefert der Aufruf `kleinstes [42,7,23]` den Wert `7`. Die Funktion darf sich auf leeren Listen beliebig verhalten.

d) **`plaetten xs`**
Erzeugt aus einer Liste von Listen von Zahlen eine Liste von Zahlen. Dabei werden die Elemente der Liste, die als Argument übergeben wird, konkateniert. Beispielsweise berechnet `plaetten [[6,28,0],[],[7,2]]` die Liste `[6,28,0,7,2]`.

Lösungsvorschlag

```
-- a
millimeter :: Int -> Int -> Int
millimeter feet inch = 305*feet + 25*inch

-- b
```

```
singletons :: Int -> [[Int]]
singletons 0 = []
singletons n = [n] : singletons (n-1)

-- c
kleinstes :: [Int] -> Int
kleinstes (x:xs) = kleinstesH x xs

kleinstesH :: Int -> [Int] -> Int
kleinstesH m []     = m
kleinstesH m (y:ys) = if (m < y) then kleinstesH m ys
                      else kleinstesH y ys

-- alternative Loesung:
kleinstes' :: [Int] -> Int
kleinstes' [x]      = x
kleinstes' (x:y:xs) = if (x < y) then kleinstes' (x:xs)
                      else kleinstes' (y:xs)

-- d
plaetten :: [[Int]] -> [Int]
plaetten [] = []
plaetten ([]:xs) = plaetten xs
plaetten ((x:xs):ys) = x : plaetten (xs:ys)
```

Abgabe zum 21.01.2014

[1] Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.