Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

9. Februar 2015

Aufgaben zur Klausur **Grundlagen der funktionalen Programmierung** im WS 2014/15 (BInf)

Zeit: 90 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 7 Seiten.

Aufgabe 1:

Entwickeln Sie eine Funktion `factors` mit folgendem Typ

$$factors :: Int \rightarrow [Int]$$

Diese Funktion soll alle Teiler einer Zahl in einer Liste sammeln. Die Liste soll aufsteigend sortiert sein. Hinweis: Es brauchen für die Lösung nur positive Zahlen betrachtet werden. Jeder Teiler soll in der Liste nur einmal vorkommen. Zum Beispiel ergibt sich für `n = 24` folgendes Resultat `[1,2,3,4,6,8,12,24]`.

.....

.....

.....

.....

.....

Entwickeln Sie eine zweite Funktion `primefactors` mit gleichem Typ wie `factors`. Diese Funktion soll die Liste aller Primfaktoren für eine Zahl berechnen. Diese Liste soll wieder sortiert sein. Die Funktion soll folgende Eigenschaft erfüllen:

$$product\ (primefactors\ n) = n$$

für `n = 24` ergibt sich also das Ergebnis `[2,2,2,3]`, für `n = 23` das Ergebnis `[23]`.

Hinweis: Implementieren Sie die Funktion mit einer rekursiven Hilfsfunktion mit einem zusätzlichen Parameter für die möglichen Teiler

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 2:

Die Implementierung der natürlichen Zahlen als Peano-Zahlen einschließlich Addition und Subtraktion hat folgende Gestalt:

```
data Nat
   = Zero
   | Succ Nat
add                    :: Nat → Nat → Nat
add Zero       n2      = n2
add (Succ n1) n2       = Succ (add n1 n2)
sub                    :: Nat → Nat → Nat
sub n1         Zero    = n1
sub (Succ n1) (Succ n2) = sub n1 n2
sub Zero       (Succ n2) = error "no negative numbers"
```

Man kann die Peano-Zahlen auf folgende Art um negative Zahlen erweitern:

```
data Intg
   = Zero
   | Succ Intg
   | Pred Intg
succ           :: Intg → Intg
succ (Pred n) = n
succ n         = Succ n
pred           :: Intg → Intg
pred (Succ n) = n
pred n         = Pred n
```

Die negativen Zahlen werden mit `Pred`-Konstruktoren dargestellt, `Pred Zero` repräsentiert die -1, `Pred (Pred Zero)` die -2 usw. Es muss die Konsistenzbedingung eingehalten werden, dass `Pred`- und `Succ`-Konstruktoren nie gemischt werden. Die Funktionen zum Zählen sind schon vorgegeben.

Erweitern Sie die Funktionen `add` und `sub` so, dass sie auf `Intg` arbeiten und beide total definiert sind. Tipp: Die Rekursion bricht bei `Zero` ab, bei positiven Zahlen muss also die Anzahl der `Succ`-Konstruktoren verringert werden, bei negativen die Anzahl der `Pred`-Konstruktoren.

$add :: Intg \to Intg \to Intg$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

$sub :: Intg \to Intg \to Intg$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 3:

Gegeben seien die folgenden rekursiven Funktionen. Diese besitzen ähnliche Strukturen.

$$
\begin{array}{ll}
sum & :: Num\ a \Rightarrow [a] \to a \\
sum\ xs & = sum'\ 0\ xs \\
\quad \textbf{where} & \\
\quad sum'\ r\ [\,] & = r \\
\quad sum'\ r\ (x : xs) & = sum'\ (r + x)\ xs \\
reverse & :: [a] \to [a] \\
reverse\ xs & = reverse'\ [\,]\ xs \\
\quad \textbf{where} & \\
\quad reverse'\ r\ [\,] & = r \\
\quad reverse'\ r\ (x : xs) & = reverse'\ (x : r)\ xs \\
mean & :: [Double] \to Double \\
mean\ xs & \\
\quad \mid l > 0 & = s\ /\ l \\
\quad \mid otherwise & = 0 \\
\quad \textbf{where} & \\
\quad\quad (s, l) & = mean'\ (0.0, 0.0)\ xs \\
\quad\quad mean'\ acc\ [\,] & = acc \\
\quad\quad mean'\ (s', l')\ (x : xs) & = (s' + x, l' + 1)
\end{array}
$$

Um solche ähnlich strukturierten Funktionen einfacher zu implementieren, gibt es die so genannten `fold`-Funktionen. `foldl` ist eine dieser Funktionen mit folgendem Typ:

$$foldl :: (b \to a \to b) \to b \to [a] \to b$$

Implementieren Sie diese `foldl`-Funktion:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Reimplementieren sie `sum` mit Hilfe von `foldl`

.....

.....

Reimplementieren sie `reverse` mit Hilfe von `foldl`

.....

.....

.....

Reimplementieren sie `mean` mit Hilfe von `foldl`

.....

.....

.....

.....

.....

.....
