Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

22. Februar 2021

Aufgaben zur Klausur **Grundlagen der funktionalen Programmierung** im WS 2020/21 (BInf, ITAS)

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 10 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 darf angenommen werden, dass die Funktion nur mit Zahlen ≥ 0 aufgerufen wird. 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]`.

Definieren Sie die Funktion mit Hilfe der vordefinierten `filter`-Funktion für Listen mit dem Typ:

```
filter :: (a -> Bool) -> [a] -> [a]
```

.....

.....

.....

.....

.....

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 aufsteigend sortiert sein, kann aber gleiche Werte mehrfach enthalten. 1 ist per Definition kein Primfaktor.

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]` und für `n = 1` das Ergebnis `[]`.

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

Aufgabe 2:

Gegeben sei die folgende Datenstruktur:

```
data Bin = E
         | O Bin
         | I Bin
  deriving Show
```

Mit dieser rekursiven Datenstruktur können Binärzahlen repräsentiert werden. Der Konstruktor `E` repräsentiert das Ende der Ziffernfolge einer Binärzahl, der Konstruktor `O` die Ziffer 0 gefolgt von den restlichen Ziffern, und der Konstruktor `I` die Ziffer 1 gefolgt von den restlichen Ziffern.

Die folgende Funktion `toBin` konvertiert eine nicht negative Zahl in diese Binärdarstellung:

```
toBin :: Int → Bin
toBin n
  | n > 0 = if even n
            then O b
            else I b
  | n == 0 = E
  | n < 0 = error "no negative numbers"
  where
    b = toBin (n `div` 2)
```

Es gilt also: `toBin 13 = I (O (I (I E)))` $(13 = 1 + 4 + 8)$

Entwickeln Sie die inverse Funktion `fromBin`, die einen `Bin`-Wert in einen `Int`-Wert zurück konvertiert, es soll also gelten: `fromBin . toBin == id`

Der Code für `fromBin`:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Die einfachste arithmetische Funktion auf natürlichen Zahlen ist eine Zahl um 1 zu erhöhen. Hierfür wird eine Funktion `incr :: Bin -> Bin` benötigt. Für alle natürlichen Zahlen n muss dann folgendes Gesetz gelten: `fromBin (incr (toBin n)) == n + 1`

Entwickeln Sie die Funktion `incr`:

.....

.....

.....

.....

.....

.....

.....

Binärzahlen werden wie Dezimalzahlen ziffernweise addiert. Dabei können Überträge entstehen, die auf die höheren Stellen übertragen werden müssen. Es gibt einige Fallunterscheidungen, da bei der Addition gleichzeitig über die Struktur zweier Zahlen verzweigt werden muss.

Wenn eine der zu addierenden Zahlen 0 ist, dann ist das Ergebnis klar. Enthalten beide Zahlen mindestens eine Ziffer, so müssen die erste Ziffer beider Zahlen addiert werden und die restlichen Stellen verarbeitet werden. Ein Übertrag entsteht nur, wenn die Ziffern $1 + 1$ addiert werden. Dann ist der Übertrag 1, es muss in diesem Fall also irgenwo eine Zahl um 1 erhöht werden. Diese Tipps führen zu einer Funktion mit 6 Fallunterscheidungen. Entwickeln Sie erst die Fallunterscheidungen, bevor sie sich an die Entwickung der einzelnen Fälle machen.

Eine Beispiel $42 + 12 = 54$:

$$
\begin{aligned}
\mathit{toBin}\ 42 &== O\ (I\ (O\ (I\ (O\ (I\ E)))))\\
\mathit{toBin}\ 12 &== O\ (O\ (I\ (I\ E)))\\
\mathit{toBin}\ 54 &== O\ (I\ (I\ (O\ (I\ (I\ E)))))
\end{aligned}
$$

Der Code für `add :: Bin -> Bin -> Bin`:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 3:

Gegeben seien die folgenden rekursiven Funktionen. Diese besitzen alle eine ähnliche Struktur.

$$
\begin{array}{ll}
prod & :: Num\ a \Rightarrow [a] \to a \\
prod\ [\,] & = 1 \\
prod\ (x : xs) & = x * prod\ xs \\
\\
concat & :: [[a]] \to [a] \\
concat\ [\,] & = [\,] \\
concat\ (xs : xss) & = xs +\!\!+ concat\ xss \\
\\
map & :: (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = [\,] \\
map\ f\ (x : xs) & = f\ x : map\ f\ xs \\
\\
andOr & :: [Bool] \to (Bool, Bool) \\
andOr\ [\,] & = (True, False) \\
andOr\ (b : bs) & = (b\ \&\&\ b1, b\ ||\ b2) \\
 & \quad \textbf{where} \\
 & \quad\ (b1, b2) = andOr\ bs
\end{array}
$$

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

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

Implementieren Sie diese `foldr`-Funktion:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Reimplementieren sie `prod` mit Hilfe von `foldr`

.....

.....

.....

Reimplementieren sie `concat` mit Hilfe von `foldr`

.....

.....

.....

Reimplementieren sie `map` mit Hilfe von `foldr`

.....

.....

.....

.....

Reimplementieren sie `andOr` mit Hilfe von `foldr`

.....

.....

.....

.....

.....

Aufgabe 4:

Es sei folgender rekursiver Datentyp gegeben:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ \ a\\ &\qquad\qquad\ \ \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

Entwickeln Sie eine Funktion `mapTree`, mit der alle Elemente eines Baumes verarbeitet werden können, ohne die Struktur des Baumes zu verändern.

$$\mathit{mapTree} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Gegeben sei eine Funktion

$$\begin{aligned}&\mathit{flatten} :: \mathit{Tree}\ a \to [a]\\ &\mathit{flatten}\ (\mathit{Leaf}\ x) = [x]\\ &\mathit{flatten}\ (\mathit{Node}\ l\ r) = \mathit{flatten}\ l +\!\!+ \mathit{flatten}\ r\end{aligned}$$

Entwickeln sie eine Funktion `reverseTree`, die den Baum so umstrukturiert, das gilt:
`flatten . reverseTree == reverse . flatten`

$$\mathit{reverseTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ a$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `foldTree`, mit der ein Baum zu einem Wert *zusammengefaltet* werden kann. Tipp: Orientieren Sie sich bei der Entwicklung der Funktion an der Signatur.

$$foldTree :: (b \to b \to b) \to (a \to b) \to Tree\ a \to b$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Nutzen Sie die Funktion `foldTree` zum Aufsummieren aller Elemente eines Baums, in dem Zahlen gespeichert sind.

$$sumTree :: Num\ a \Rightarrow Tree\ a \to a$$

. . . . . .

Nutzen Sie die Funktion `foldTree` zur Reimplementierung der `flatten`-Operation.

. . . . . .

. . . . . .

. . . . . .

Nutzen Sie die Funktion `foldTree` zur Reimplementierung der `mapTree`-Operation.

. . . . . .

. . . . . .

. . . . . .