Alp 1 - WS 02/03 (Tutorium: Till Zoppke) Do_12
Monika Budde / Emrah Somay
Uebung 7
****

Aufg. 1

a) Eine geschlossene Formel für

```
f :: Int -> Int
f 0 = 1                                 -- (1)  Rekursionsverankerung
f n = sum [f x | x <- [0..(n-1)] ]      -- (2)
```

als Funktion auf den natürlichen Zahlen ($f : \mathbb{N} \to \mathbb{N}$) ist $f(n) = 2^{n \dot{-} 1}$.

Beweis:

Wir zeigen zunächst durch Induktion über die Struktur des Arguments von `sum`, daß `sum ys = sum [f i | i <- [0..n]]` für jede Funktion $f: \mathbb{N} \to \mathbb{N}$ die Funktion $\sum_{i=0,\dots,n} f(i)$ berechnet. Dazu verwenden wir die vereinfachte Beschreibung von `sum` aus dem Tutorium:

```
sum :: [Int] -> Int
sum [] = 0                       -- (3)  Rekursionsverankerung
sum (x:xs) = x + sum xs          -- (4)
```

Induktionsanfang (`sum`):

Sei `ys = [] = [f i | i <- [0..(-1)]]`. Dann gilt nach (3):

`sum ys` $= 0 = \sum_{i=0,\dots,-1} f(i)$ (= ‘die leere Summe’).

Induktionsschritt (`sum`):

Sei die Behauptung über die Summenfunktion für eine Liste `xs` der Länge n mit `xs = [f i | i <- [0..n]]` bereits bewiesen (Induktionsvoraussetzung). Dann gilt für jede Liste `ys` mit `ys = ( f(n+1) : xs)`:

`sum ys = sum ( f(n+1) : xs) = f(n+1) + sum xs` (nach (4))

$= f(n+1) + \sum_{i=0,\dots,n} f(i)$ (nach Induktionsvorauss.)

$= \sum_{i=0,\dots,n+1} f(i)$

Also gilt die Behauptung über die Summenfunktion für alle Listen der Form `ys = [f i | i <- [0..n]]`.

Jetzt können wir die eigentliche Behauptung zeigen (durch Induktion über n):

Induktionsanfang:

Sei $n = 0$. Dann gilt nach (1): $f(n) = f(0) = 1 = 2^0 = 2^{0 \dot{-} 1}$.

Induktionsschritt:

Sei die Behauptung für ein $n_0 \in \mathbb{N}$ und alle $n \in \mathbb{N}$, $n \leq n_0$ bereits bewiesen (Induktionsvoraussetzung). Dann gilt:

$f(n+1) = \sum_{i=0,\dots,n} f(i) = \sum_{i=0,\dots,n} 2^{i \dot{-} 1} = 1 + \sum_{i=1,\dots,n} 2^{i-1}$ (nach Induktionsvoraussetzung)

$= 1 + \sum_{i=0,\dots,n-1} 2^i = 1 + 2^n - 1$ (bekannte Formel; vgl. Binärdarstellung)

$= 2^n = 2^{(n+1)-1} = 2^{(n+1) \dot{-} 1}$ □

b) eine geschlossene verbale Beschreibung für

```
f :: String -> String
f [] = []                                       -- (1)
f (x:xs) = if x == 'a' then fs else x:fs        -- (2)
    where fs = f xs                             -- (3)
```

ist „entferne alle Vorkommen von `'a'` in dem Argument-String `ys` und gib das Resultat aus“.

Beweis (durch Induktion über die Struktur von `ys`):

Induktionsanfang:
Sei `ys = []`. Dann gilt nach (1):
`f [] = []` = ‘die leere Liste nach dem Entfernen aller Vorkommen von `'a'` ’

Induktionsschritt:
Sei die Behauptung für eine Liste `xs` bereits bewiesen (Induktionsvoraussetzung) und sei `ys = x:xs`. Dann gilt:

Fall 1: `x == 'a'`, also
`f (x:xs) = fs = f xs` (nach (2) und (3))
= ‘`xs` nach dem Entfernen aller Vorkommen von `'a'` ’ (nach Induktionsvoraussetzung)
= ‘`ys` nach dem Entfernen aller Vorkommen von `'a'` ’

Fall 2: `x /= 'a'`, also
`f (x:xs) = x:(f xs)` (nach (2) und (3))
= ‘`x` angehängt an `xs` nach dem Entfernen aller Vorkommen von `'a'` in `xs` ’
(nach Induktionsvoraussetzung)
= ‘`ys` nach dem Entfernen aller Vorkommen von `'a'` ’ □

****

Aufg. 2: strukturelle Induktion über Listen

a) *Behauptung:* length (reverse xs) = length xs

Beweis:

Zunächst stellen wir die benötigten Definitionen zusammen (die Definition des Listenkonstruktors „:“ und die der Addition sowie die Gesetze zur Addition benutzen wir stillschweigend):

Def.:
```
length :: [a] -> Int
length [] = 0                          -- (lg1)
length(_:xs) = 1 + length xs           -- (lg2)

reverse :: [a] -> [a]
reverse [] = []                        -- (r1)
reverse (x:xs) = (reverse xs) ++ [x]   -- (r2)

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys                          -- (cons1)
(x:xs) ++ ys = x : (xs ++ ys)          -- (cons2)
```

Wir beweisen zunächst (durch strukturelle Induktion), daß für alle Listen xs, ys (vom gleichen Typ) gilt:

(*) length (xs ++ ys) = (length xs) + (length ys)

Induktionsanfang:
Sei ys irgendeine Liste. Dann gilt für xs = [] :

length (xs ++ ys) = length ([] ++ ys) = length (ys)	(nach (cons1))
= 0 + (length ys) = (length []) + (length ys)	(nach (lg 1), rückwärts)
= (length xs) + (length ys)	

Induktionsschritt:
Sei die Behauptung length (zs ++ ys) = (length zs) + (length ys) für eine Liste zs und alle Listen ys vom Typ der Liste zs bewiesen (Induktionsvoraussetzung). Dann folgt für xs = (z:zs):

length (xs ++ ys) = length ((z:zs) ++ ys) = length (z: (zs ++ ys))	(nach (cons2))
= 1 + length (zs ++ ys)	(nach (lg2))
= 1 + (length zs) + (length ys)	(nach Induktionsvorauss.)
= (length(z:zs)) + (length ys)	(nach (lg2), rückwärts)

Jetzt können wir durch strukturelle Induktion beweisen: length (reverse xs) = length xs.

Induktionsanfang:

xs = [] ⇒ length (reverse xs) = length (reverse []) = length ([])	(nach (r1))
= length xs	

Induktionsschritt:
Sei die Behauptung für eine Liste ys bewiesen. Sei xs eine Liste mit xs = (y:ys). Dann gilt:

length (reverse xs) = length (reverse (y:ys)) = length((reverse ys) ++ [y])	(nach (r2))
= (length(reverse ys)) + (length [y])	(nach (*))
= (length ys) + (length [y])	(nach Ind.vorauss.)
= (length [y]) + (length ys) = length ([y] ++ ys)	(nach (*))
= length((y:[]) ++ ys) = length (y : ([] ++ ys))	(nach (cons2))
= length (y:ys)	(nach (cons1))
= length xs	□

b) *Behauptung:* entf xs = entf (entf xs),
wobei gilt:

```
elem2 :: Char -> String -> Bool
elem2 _ []        = False
elem2 x (y:ys)    = x == y || (elem2 x ys)

entf :: String -> String
entf [] = []                                  -- (entf3)
entf (c:cs)
    |elem2 c "aeiouäöüAEIOUÄÖÜ" = entf cs     -- (entf2), c Vokal
    |otherwise  = c : entf cs                 -- (entf3)
```

Beweis (durch strukturelle Induktion):
Induktionsanfang:
xs = [] ⇒ entf (entf []) = entf [] (nach (entf1))

Induktionsschritt:
Sei die Behauptung für eine Liste ys bewiesen (Induktionsvoraussetzung). Dann folgt für jede Liste xs = (y:ys) :
Fall 1: y ist Vokal, d.h. elem2 y "aeiouäöüAEIOUÄÖÜ", also
entf (entf xs) = entf (entf (y:ys)) = entf (entf ys) (nach (entf2))
= entf ys (nach Induktionsvoraussetzung)
Fall 2: y ist kein Vokal, d.h. der Fall otherwise ist anzuwenden (zweimal), also
entf (entf xs) = entf (entf(y:ys)) = entf (y: entf ys) (nach (entf3))
= y:entf (entf ys) (nach (entf3))
= y:entf ys (nach Induktionsvoraussetzung)
= entf (y:ys) (nach (entf3), rückwärts)
= entf xs □

****

Aufg. 3
a) Haskell-Programm: berechnete Funktion
```
f :: Int -> Int
f 0 = 1                -- (1)  Rekursionsverankerung
f n = 4 * f(n-2)       -- (2)
```

Dieses Programm terminiert für alle geraden natürlichen Zahlen (einschließlich 0). Für eine ungerade natürliche Zahl und für negative Zahlen terminiert das Programm nicht, d.h. die Rekursionsverankerung wird nie erreicht: Für negative Zahlen n ist dies offensichtlich ($n-2 = -|n| -2 < n < 0$).
Für ungerade natürliche Zahlen n gilt: Es gibt eine natürliche Zahl m mit $n = 2*m + 1$. Damit folgt:
$\mathrm{f}\, n = \mathrm{f}\,(2*m + 1) = \underbrace{(4 * \ldots * 4)}_{m\text{-mal}} * \mathrm{f}(1) = (\prod_{i = 1, \ldots m} 4\,) * \mathrm{f}(1) = (\prod_{i = 1, \ldots m+1} 4\,) * \mathrm{f}(-1)$
Also terminiert f auch für ungerade natürliche Zahlen nicht.

Das Programm berechnet die partielle Funktion $\mathrm{f} : \mathbb{N} \to \mathbb{N}$, $\mathrm{f}\,(n) = 4^{n/2}$, falls n gerade, undefiniert sonst.
Beweis (durch Induktion über n):
Induktionsanfang:
Sei $n = 0$. Dann gilt nach (1): $\mathrm{f}\,(n) = \mathrm{f}\,(0) = 1 = 4^0$.

Induktionsschritt:
Sei die Behauptung für eine gerade natürliche Zahl n bereits bewiesen (Induktionsvoraussetzung).
Dann ist die nächstgrößere gerade natürliche Zahl $n + 2$. Für $n + 2$ gilt:
$\mathrm{f}\,(n + 2) = 4 * \mathrm{f}\,(n) = 4 * 4^{n/2}$ (nach Induktionsvoraussetzung)
$= 4^{1+n/2} = 4^{(n + 1)/2}$ □

b) Haskell-Funktion: Anfangsstück eines Strings

Programm und Definitionsbereich: s. Listing

Korrektheitsbeweis (durch Induktion über die Struktur des 2. Arguments, mit den Bezeichnungen und Zeilennummern aus dem Listing):

Seien x und xs ein Argumentpaar, für das anfVor definiert ist (d.h.: wenn xs eine unendliche Liste ist, dann kommt x in xs vor).

Induktionsanfang:

Fall 1: Sei xs == []. Dann gilt: anfVor x xs = anfVor x [] = [] für jedes Element x (nach (1)).

Fall 2: Sei xs /= [] , also xs = y:ys für einen String ys und ein Element y, und x == y. Dann gilt: anfVor x xs = anfVor x (y:ys) = [x] (nach 2).

Induktionsschritt:

Sei die Korrektheit von anfVor für einen String ys und ein Element x bereits bewiesen (Induktionsvoraussetzung). Dann gilt für xs = (y:ys) :

Fall 1: x == y. Dann gilt: anfVor x xs = anfVor x (y:ys) = [x] (nach (2); ys irrelevant).

Fall 2: x /= y, x kommt in ys vor: anfVor x xs = anfVor x (y:ys) = y: (anfVor x ys) (nach (3)). Dies ist korrekt, da anfVor x ys nach Induktionsvoraussetzung korrekt ist.

Fall 3: x /= y, x kommt in ys nicht vor: anfVor x xs = [] (nach (4)). □