Algorithmen und Programmieren 1

Funktionale Programmierung

WS2011/2012 Nachklausur Musterlösung

Julian Fleischer, Terese Haimberger, Nicolas Lehmann,
Christopher Pockrandt, Alexander Steen

24.03.2012

Aufgabe 1: 15 Punkte

Schreiben Sie eine Funktion `tiefeKonst:: Ausdruck -> Int`, die die Tiefe der am tiefsten geschachtelten Konstanten in einem arithmetischen Ausdruck bestimmt. Wenn der Ausdruck keine Konstanten enthält, soll das Ergebnis `-1` sein. Der Datentyp für arithmetische Ausdrücke ist folgendermaßen definiert:

```
data Ausdruck = Konst Integer | Var String | Plus Ausdruck Ausdruck
              | Mal Ausdruck Ausdruck deriving (Eq, Show)
```

Beispiele:

```
tiefeKonst (Konst 5)                                  = 0
tiefeKonst (Plus (Var "a") (Konst 5))                 = 1
tiefeKonst (Plus (Var "a") (Var "b"))                 = (-1)
tiefeKonst (Plus (Mal (Var "x") (Konst 2)) (Konst 5)) = 2
```

Bei allen Funktionen ist wie immer eine Typdeklaration anzugeben.

Lösung zu Aufgabe 1

```
-- Bestimmt die Tiefe der am tiefsten geschachtelten Konstanten in einem Ausdruck.
tiefeKonst :: Ausdruck -> Int
tiefeKonst e                                          -- e = Ausdruck
           | tiefeListe == [] = -1                    -- -1, falls kein Konst im Ausdruck
           | otherwise        = maximum (tiefeListe)  -- bestimmt maximale Tiefe sonst
             where
                tiefeListe = expr2list e 0            -- Aufruf der Hilfsfunktion

-- Macht aus einem Ausdruck eine Liste, in der die Tiefe aller im Ausdruck vorkommenden
-- Konstanten gespeichert werden.
expr2list :: Ausdruck -> Int -> [Int]
expr2list e t                                                  -- e = Ausdruck, t = Tiefe
       | e == (Var _)    = []                                  -- wird nicht berücksichtigt
       | e == (Konst _)  = [t]                                 -- Tiefe wird gespeichert
       | e == (Plus x y) = expr2list x (t+1) ++ expr2list y (t+1)  -- wird nicht berücksichtigt
       | e == (Mal x y)  = expr2list x (t+1) ++ expr2list y (t+1)  -- wird nicht berücksichtigt
```

Aufgabe 2: 15 Punkte

Welche Variablen kommen im Ausdruck $\lambda x.((\lambda x.\lambda a.(ax(bx)yx))(\lambda b.xb))$ gebunden vor?
Welche Variablen kommen frei vor (bezogen auf den gesamten Ausdruck)?
Reduzieren Sie den Ausdruck so weit wie möglich.

Lösung zu Aufgabe 2

$\lambda x.((\lambda x.\lambda a.(ax(bx)yx))(\lambda b.xb))$

Gebundene Variablen: x, a, b
Freie Variablen: b, y

β-Reduktion:
$\lambda x.((\lambda x.\lambda a.(ax(bx)yx))(\lambda b.xb))$
$\Longrightarrow^{\beta} \lambda x.((\lambda a.(a(\lambda b.xb)(b(\lambda b.xb))y(\lambda b.xb))))$

Aufgabe 3: 20 Punkte

Schreiben Sie ein Haskell-Programm `vielfachheit::[Int] -> [Int]`, das für eine Eingabeliste `x` von nichtnegativen Zahlen zählt, wie oft jede Zahl in ihr enthalten ist. Das Ergebnis ist eine Liste `b`, wobei `b!!i` angibt, wie oft `i` in `x` vorkommt.
Beispiel: `vielfachheit [1,3,5,3,1] = [0,2,0,2,0,1]`
Verwenden Sie Kommentare zur Erläuterung Ihres Programms. Bei allen Funktionen ist wie immer eine Typdeklaration anzugeben.

Analysieren Sie die asymptotische Laufzeit Ihres Programms. Überlegen Sie dazu, von welchen Parametern der Eingabe die Laufzeit abhängt, und geben Sie dann eine obere Schranke für die Laufzeit in O-Notation an.

Sie dürfen die Funktion `sort` aus dem Modul `Data.List` verwenden, die eine Folge der Länge n mit Laufzeit $O(n \cdot log(n))$ sortiert.

Lösung zu Aufgabe 3

```
-- Zählt das Vorkommen jeder Zahl x aus dem Intervall [0;maximum xs] und speichert
-- dieses in einer Ergebnisliste.
vielfachheit :: [Int] -> [Int]
vielfachheit xs = [(length (filter (==x) xs)) | x <- [0..(maximum xs)]]
```

Laufzeit:

$$\Longrightarrow O(\underbrace{n}_{\texttt{length}} \cdot \underbrace{n}_{\texttt{filter}} \cdot (maximum\, xs + 1)) = O(n^2 \cdot (maximum\, xs + 1))$$

Alternative Lösung zu Aufgabe 3

```
-- Zählt, wie oft eine Zahl in der Eingabeliste enthalten ist.
-- Die Funktion vielfachheit ruft eine Hilfsfunktion auf.
vielfachheit:: [Int] -> [Int]
vielfachheit [] = []
vielfachheit xs = helper xs (maximum xs) []                -- Aufruf der Hilfsfunktion

-- Die Hilfsfunktion baut mit Hilfe einer gegen (-1) strebenden Positionsvariablen eine
-- Ergebnisliste rückwärts zusammen.
helper :: [Int] -> Int -> [Int] -> [Int]
helper list (-1) resultlist  = resultlist                  -- Anker: Ergebnisliste ausgeben
helper list position resultlist                            -- Ergebnisliste zusammenbauen
       | (filter (==position) list) == [] = helper (list) (position-1) (0:resultlist)
       | otherwise                        = helper (filter (/=position) list) (position-1)
                                              ((length (filter (==position) list)):resultlist)
```

Laufzeiten:

$$\Longrightarrow O(\underbrace{n \cdot n \cdot (maximum\, xs + 1)}_{\texttt{helper}} + \underbrace{n}_{\texttt{maximum}}) = O(n^2 \cdot (maximum\, xs + 1))$$

Aufgabe 4: 20 Punkte

Welche Formel muss man für x einsetzen, damit die Gleichung

```
drop m . drop n = drop (x)
```

für *alle* m und n (auch für negative Werte) gilt?

Beweisen Sie die Gleichung dann durch strukturelle Induktion (oder durch eine andere Methode Ihrer Wahl) unter Verwendung der folgenden Definitionen:

```
     drop :: Int -> [a] -> [a]
(d1) drop n xs | n <= 0 = xs
(d2) drop _ []          = []
(d3) drop n (_:xs)      = drop (n-1) xs
```

Lösung zu Aufgabe 4

Mögliche Lösungen für `x`:

```
x = ((max m 0) + (max n 0))

x = fkt m n
       | m <= 0 && n <= 0 = 0
       | m >  0 && n <= 0 = m
       | m <= 0 && n >  0 = n
       | m >  0 && n >  0 = m+n
```

Es folgt ein Beweis mit struktureller Induktion über xs.

zu zeigen: `drop m . drop n = drop ((max m 0) + (max n 0))`

$\Longleftrightarrow$ `drop m (drop n xs) = drop ((max m 0) + (max n 0)) xs`

Induktionsbasis: `xs = []`

```
drop m (drop n []) = drop ((max m 0) + (max n 0)) []
⟺d2 drop m [] = drop ((max m 0) + (max n 0)) []
⟺d2 [] = drop ((max m 0) + (max n 0)) []
⟺d2 [] = []
```

Induktionsvoraussetzung: Für eine Liste `xs` mit beliebiger, aber fester Länge gilt:
`drop m (drop n xs) = drop ((max m 0) + (max n 0)) xs`

Induktionsbehauptung: Dann gilt auch:
`drop m (drop n (x:xs)) = drop ((max m 0) + (max n 0)) (x:xs)`

Induktionsschritt: `xs` $\longrightarrow$ `(x:xs)`

`drop m (drop n (x:xs)) = drop ((max m 0) + (max n 0)) (x:xs)`

Fall 1: $m <= 0, n <= 0$
```
drop m (drop n (x:xs)) = drop 0 (x:xs)
⟺d1 drop m (drop n (x:xs)) = drop 0 (x:xs)
⟺d1 drop m (x:xs) = drop 0 (x:xs)
⟺d1 (x:xs) = drop 0 (x:xs)
⟺d1 (x:xs) = (x:xs)
```

Fall 2: $m > 0, n <= 0$
```
drop m (drop n (x:xs)) = drop m (x:xs)
⟺d1 drop m (x:xs) = drop m (x:xs)
```

Fall 3: $m <= 0, n > 0$
```
drop m (drop n (x:xs)) = drop n (x:xs)
⟺d1 drop n (x:xs) = drop n (x:xs)
```

Fall 4: $m > 0, n > 0$
```
drop m (drop n (x:xs)) = drop (m+n) (x:xs)
⟺d3 drop m (drop (n-1) xs) = drop (m+n) (x:xs)
⟺d3 drop m (drop (n-1) xs) = drop (m+n)-1 xs
⟺assoz. drop m (drop (n-1) xs) = drop m+(n-1) xs
⟺IV drop m+(n-1) xs = drop m+(n-1) xs
```

Es wurde mit struktureller Induktion gezeigt, dass die Behauptung gilt.

Aufgabe 5: 15 Punkte

Die folgenden Funktionen sind definiert:

```
rotate xs (y:ys) rys = if   xs == []
                       then y:rys
                       else (head xs):(rotate (tail xs) ys (y:rys))

folge n = n:(folge (n+1))
```

Beschreiben Sie, wie Haskell den Ausdruck `take 6 (rotate [2,4,5] (folge 5) (folge 9))` Schritt für Schritt auswertet. Achten Sie darauf, dass Funktionen nicht ausgewertet werden, bevor ihr Wert benötigt wird.

Lösung zu Aufgabe 5

```
take 4 (rotate [2,4,5] (folge 5) (folge 9))
take 4 (rotate [2,4,5] (5:(folge 6)) (folge 9))
take 4 (2:(rotate [4,5] (folge 6) (5:(folge 9))))
2: take 3 (rotate [4,5] (folge 6) (5:(folge 9)))
2: take 3 (rotate [4,5] (6:(folge 7)) (5:(folge 9)))
2: take 3 (4:(rotate [5] (folge 7) (6:5:(folge 9))))
2:4: take 2 (rotate [5] (folge 7) (6:5:(folge 9)))
2:4: take 2 (rotate [5] (7:(folge 8)) (6:5:(folge 9)))
2:4: take 2 (5:(rotate [] (folge 8) (7:6:5:(folge 9))))
2:4:5: take 1 (rotate [] (folge 8) (7:6:5:(folge 9)))
2:4:5: take 1 (rotate [] (8:(folge 9)) (7:6:5:(folge 9)))
2:4:5: take 1 (8:7:6:5:(folge 9))
2:4:5:8: take 0 (7:6:5:(folge 9))
2:4:5:8: []
[2,4,5,8]
```

Aufgabe 6: 5 Punkte

Bestimmen Sie den allgemeinsten Typ, den die Funktion `f` haben kann.

```
f a b c
  | a b       = a b
  | otherwise = a c
```

Begründen Sie Schritt für Schritt, wie Sie zu Ihren Schlussfolgerungen kommen.

Lösung zu Aufgabe 6

- Aus `a b` im ersten Guard kann man schließen, dass die Funktion `a` eine Funktion vom Typ `(x -> Bool)` ist, wobei `x` der Typ von `b` ist.
  `f::(x -> Bool) -> (x) -> (?) -> (?)`
- Da das Ergebnis des ersten Guards auch `a b` ist, ist der Ergebnistyp der Funktion `Bool`
  `f::(x -> Bool) -> (x) -> (?) -> Bool`
- Da der Ergebnistyp der Funktion in jedem Guard gleich sein muss, hat die Funktion `a c` im zweiten Guard den selben Ergebnistyp wie die Funktion `a b`. Wir können direkt daraus folgern, dass `c` den selben Typ hat wie `b`, da beide Parameter der Funktion `a` sind.
  `f::(x -> Bool) -> (x) -> (x) -> Bool`