Funktionale Programmierung – WS 2014/15

Musterlösung zur 1. Klausur am 17.02.2015

Die nachfolgende Musterlösung stellt jeweils nur einen von möglicherweise mehreren Lösungswegen vor. Die Kommentare sind bereits relativ kurz gefasst, aber man könnte es aber noch weiter verkürzen, indem Stichpunkte an Stelle von ganzen Sätzen verwendet werden. Passagen, an denen Ideen zum besseren Verständnis noch genauer beschrieben werden, sind als Zusatzkommentar gekennzeichnet. Solche Zusatzkommentare sind also für das Erreichen der vollen Punktzahl nicht relevant.

Da ich übesehen hatte, dass die Teilaufgaben 1.a und 1.b nur in der Vorlesung besprochen wurden, aber in den Übungen nicht vorkamen, wurden noch vier weitere Punkte (also insgesamt sieben) als Zusatzpunkte deklariert. Damit reichen jetzt 18 Punkte zum Bestehen der Klausur.

Die erreichbaren Punkte sind bereits in der Kopfzeile der Aufgaben auf die einzelnen Teilaufgaben aufgegliedert worden. Um die Feinverteilung der Punkte noch transparenter zu machen, wird bei der Musterlösung an einigen Stellen angezeigt, auf welche Ideen, Ansätze und Teilschritte man bereits Punkte bekommen kann, wenn die Lösung der Aufgabe nicht vollständig erbracht wurde. Dabei steht das Symbol $\odot$ für einen halben und das Symbol $\bullet$ für einen ganzen Punkt.

Aufgabe 1: Algebraische Typen und Typklassen $2+1+1+3$ Punkte

Gegeben sind die folgenden Datentypen für Variablen und arithmetische Ausdrücke:

```
data Variable = U | V | W | X | Y | Z deriving Eq, Ord, Show
data Exp = Var Variable | Add Exp Exp | Mult Exp Exp
```

a) Definieren Sie `Exp` als Exemplar der Klasse `Show` wobei die `show` Funktion den Haskell-Ausdruck als üblichen arithmetischen Term wie z.B. $(X * (Y + Z))$ darstellen soll.

b) Definieren Sie `Exp` als Exemplar der Klasse `Eq` wobei der Operator `==` alle Gleichheiten berücksichtigen soll, die sich aus Anwendungen des Kommutativgesetzes ergeben (Assoziativ- und Distributivgesetz sollen nicht berücksichtigt werden), d.h. für die Ausdrücke der Terme $(X * (Y + Z))$ und $((Z + Y) * X)$ soll Gleichheit und für die Ausdrücke der Terme $(X + (Y + Z))$ und $((X + Y) + Z)$ soll Ungleichheit gelten.

c) Definieren Sie eine Funktion `eval::(Variable->Int)->Exp->Int` zur Termauswertung.

d) Definieren Sie eine Funktion `varsWithDepth::Exp->[(Variable,Int)]` mit der die Liste aller in einem Ausdruck auftretenden Variablen mit der jeweiligen Tiefe im Syntaxbaum erzeugt wird. Nutzen Sie diese Funktion sowie die Listenfunktionen `maximum`, `minimum` und List-Comprehension, um Funktionen `minVar, minVarOfMaxD::Exp->Variable` zu definieren, mit denen die minimale im Ausdruck vorkommende Variable bzw. die minimale Variable unter allen Variablen maximaler Tiefe bestimmt werden (mimimale Variable bezieht sich auf die Ordnungsrelation `<=` des Typs `Variable`).

Lösung:

```
instance Show Exp where
    show (Var x)          = show x -- Typ Variable leitet Show ab
    show (Add e1 e2)      = "("++(show e1)++"+"++(show e2)++")"
    show (Mult e1 e2 )    = "("++(show e1)++"*"++(show e2)++")"

instance Eq Exp where
    (Var x)==(Var y)            = x==y -- Typ Variable leitet Eq ab
    (Add e1 e2)==(Add e3 e4)    = (e1==e3 && e2==e4) || (e1==e4 && e2==e3)
    (Mult e1 e2)==(Mult e3 e4)  = (e1==e3 && e2==e4) || (e1==e4 && e2==e3)
    _ == _                      = False -- alle anderen Faelle

 eal f (Var x)        = f x
 eval f (Add e1 e2)   = eval f e1 + eval f e2
 eval f (Mult e1 e2)  = eval f e1 * eval f e2

help::Int->Exp->[(Variable,Int)] -- Hilfsfunktion fuer varsWithDepth
 help n (Var x)       = [(x,n)]
 help n (Add e1 e2)   = help (n+1) e1 ++ help (n+1) e2
 help n (Mult e1 e2)  = help (n+1) e1 ++ help (n+1) e2

 varsWithDepth e  = help 0 e
 minVar e         = minimum[fst pair| pair<-(varsWithDepth e)]
 minVarOfMaxD e   = minimum[fst pair| pair<-(varsWithDepth e), snd pair==maxD]
                    where maxD= maximum[snd pair| pair<-(varsWithDepth e)]
```

Aufgabe 2: **Strukturelle Induktion** 5 **Punkte**

a) Nutzen Sie die Definitionen der Funktionen `map::(a->b)->[a]->[b]` und `(++)::[a]->[a]->[a]` , um mit struktureller Induktion zu beweisen, dass für beliebige `xs,ys::a` und `f::a->b` die Listen `map f (xs++ys)` und `map f xs ++ map f ys` identisch sind!

```
map f [ ]          = [ ]                    -- (1)
map f (x:xs)       = f x :  map f xs        -- (2)
 (++) [ ] ys       = ys                     -- (3)
 (++) (x:xs) ys    = x:(xs++ys)             -- (4)
```

Lösung: Strukturelle Induktion nach `n=length xs` , `ys` ist eine beliebige Liste.

Zusatzkommentar: Die verwendete Regel wird jeweils über dem Gleicheitszeichen notier Das Gleihheitszeichen wird für die Auswertung von Teilschritten sowohl von links nach rechts als auch von rechts nach links verwendet.

Induktionsanfang: $n = 0$, d.h. `xs = [ ]`

`map f ([ ]++ys)` $\overset{(3)}{=}$ `map f (ys)`

$\overset{(3)}{=}$ `[ ]++ map f (ys)`

$\overset{(1)}{=}$ `map [ ] ++ map f (ys)`

Induktionsschritt: $n \to n + 1$, d.h. IV:wahr für `xs` $\Longrightarrow$ IB: wahr für `x:xs`

`map f ((x:xs)++ys)` $\overset{(4)}{=}$ `map f (x:(xs++ys)`

$\overset{(2)}{=}$ `(f x):map f (xs++ys)`

$\overset{IV}{=}$ `(f x):(map f xs++ map f ys)`

$\overset{(4)}{=}$ `(f x:map f xs)++ map f ys`

$\overset{(2)}{=}$ `(map f (x:xs))++ map f ys`

Aufgabe 3: Auswertung und Laufzeit $1 + 4 + 2$ **Punkte +** 3 **Zusatzpunkte**

Betrachten Sie die Funktionen `list1, list2::Int->[Int]` mit

```
list1 n = [x| x<-[n..2*n], y<-[x+1..4*n], (mod y x)==0]
list2 n = [x| x<-[n..2*n], y<-[x+1..4*n], (mod y x)>0]
```

Die Laufzeiten der folgenden Funktionen sollen durch die Anzahl der ausgeführten Vergleichsoperationen `>` und `==` in Abhängigkeit vom Parameter `n` bestimmt werden. Die Antworten müssen kurz begründet werden.

a) Welche Laufzeit hat die Funktion `head.list1` (genaue Bestimmung)?
b) Bestimmen Sie möglichst gute obere Schranken für die asymptotische Laufzeit der Funktionen

```
list3 n = [x| x<-list1 n, z<-[n..2*n], x==z]

list4 n = [x| x<-list2 n, z<-[n..2*n], x==z]
```

c) Weisen Sie untere Schranken für die Funktionen aus b) nach, so dass asymptotisch scharfe Schranken der Form $\Theta(f(n))$ entstehen.

d) Welches Ergebnis und welche Laufzeit, möglichst in der Form $\Theta(f(n))$, hat die Funktion `g n = head[x| x<-list1 (n+1), y<-list1 n, x==y]` ?

Lösung:
a) Bei der Auswertung `(head.list1)n` wird `==` genau n Mal aufgerufen $\odot$:
Für $x = n$ sind die Werte $y = n+1$ bis $y = 2n-1$ nicht durch x teilbar. Mit $y = 2n$ wird die Bedingung `(mod y x)==0` zum ersten Mal erfüllt und n als Kopfelement in `list1 n` aufgenommen $\odot$.

b) Vorüberlegung: Zur Erzeugung von `list1 n` sind $O(n^2)$ Gleichheitsabfragen ausreichend $\odot$ (Zusatzkommentar: für jeden der $n+1$ Werte von x höchstens $3n$ Werte von y), aber die Länge von `list1 n` ist nur $O(n)$, denn für jedes x gibt es höchstens drei y-Werte, die ein Vielfaches von x sind $\odot$. Dagegen liegen bei `list2 n` Laufzeit und Listenlänge in $= O(n^2)$ $\bullet$.
Die Laufzeit von `list3 n = [x| x<-list1 n, z<-[n..2*n], x==z]` ist in 0(n^2) $\odot$, denn `list1 n` wird in $O(n^2)$ Zeit erzeugt und die äußere Liste erfordert auch höchstens 0(n) $\cdot\, n$ Vergleiche $\odot$.
Die Laufzeit von `list4 n = [x| x<-list2 n, z<-[n..2*n], x==z]` ist in 0(n^3) $\odot$, denn `list2 n` wird in $O(n^2)$ Zeit erzeugt und die äußere Liste erfordet höchstens 0(n^2)$\cdot n$ Vergleiche $\odot$.

c) Die Laufzeit von `list3 n` ist auch in $\Omega(n^2)$, weil schon die Erzeugung von `list1 n` mindestens $n \cdot 2n$ Vergleiche erfordert $\bullet$.
Die Laufzeit von `list4 n` ist auch in $\Omega(n^3)$, weil die Länge von `list2 n` in $\Omega(n^2)$ liegt (für jedes x erfüllen $y = x+1$ bis $y = 2x+1$ die Bedingung `(mod y x)>0` $\odot$) und somit für die äußere Liste mindestens $\Omega(n^2) \cdot n$ Vergleiche erforderlich sind $\bullet$.

d) Die Ergebnis von `g n = head[x| x<-list1 (n+1), y<-list1 n, x==y]` ist der Wert `n+1` $\bullet$ und die Laufzeit liegt in $\Theta(n)$ $\odot$. Dabei folgt $\Omega(n)$ schon aus a) $\odot$. Für die obere Schranke beobachten wir zuerst, dass das Kopfelement von `list1 (n+1)` nach $n+1$ Vergleichen vorliegt. Bei `y<-list1 n` tritt zuerst dreimal der Wert n auf, danach aber der Wert $n+1$. Bis zum ersten Auftreten von $n+1$ werden beim Aufruf von `list1 n` insgesamt $3n+n+1$ Werte von y überprüft ($3n$ für $x = n$ und $n+1$ für $x = n+1$) - auch das ist eine lineare Anzahl $\bullet$.

Aufgabe 4: **λ-Kalkül** $3+4$ **Punkte**

a) Reduzieren Sie den λ-Ausdruck $E = (\lambda x\, y.x\, y\, y)(\lambda x.y\, a)a\, b$ so weit wie möglich.

Lösung:

$$
\begin{array}{rll}
(\lambda x\, y.x\, y\, y)(\lambda x.y\, a)a\, b & = ((\lambda x.(\lambda y.x\, y\, y))(\lambda x.y\, a))a\, b & \\
& \equiv ((\lambda x.(\lambda z.x\, z\, z))(\lambda x.y\, a))a\, b & \alpha\text{-Konversion} \\
& \equiv (\lambda z.(\lambda x.y\, a)z\, z))a\, b & \beta\text{-Reduktion} \\
& \equiv ((\lambda x.y\, a)a\, a)b & \beta\text{-Reduktion} \\
& . \equiv ((y\, a)a)b & \beta\text{-Reduktion}
\end{array}
$$

b) Formen Sie den λ-Ausdruck $(\lambda x\, y.x)(\lambda z.y)y$ in einen äquivalenten CL-Ausdruck um. Zeigen Sie in den einzelnen Umformungsschritten an, welche der folgenden Umformungsregeln jeweils angewendet wurde.

$T[\ ] : \text{Expr} \longrightarrow \text{CLExpr}$

(a) $T[x] = x$

(b) $T[E_1 E_2] = T[E_1]\, T[E_2]$

(c) $T[\lambda x.E] = abs(x, T[E])$ wobei

$abs : \text{Var} \times \text{CLExpr} \longrightarrow \text{CLExpr}$

(1) $abs(x,y) = \begin{cases} \text{I} & \text{falls x=y} \\ \text{K y} & \text{sonst} \end{cases}$

(2) $abs(x,E) = K\, E$ falls $x \notin FV(E)$

(3) $abs(x, E_1 E_2) = S\,(abs(x,E_1))\,(abs(x,E_2))$ falls $x \in FV(E_1 E_2)$

Lösung:

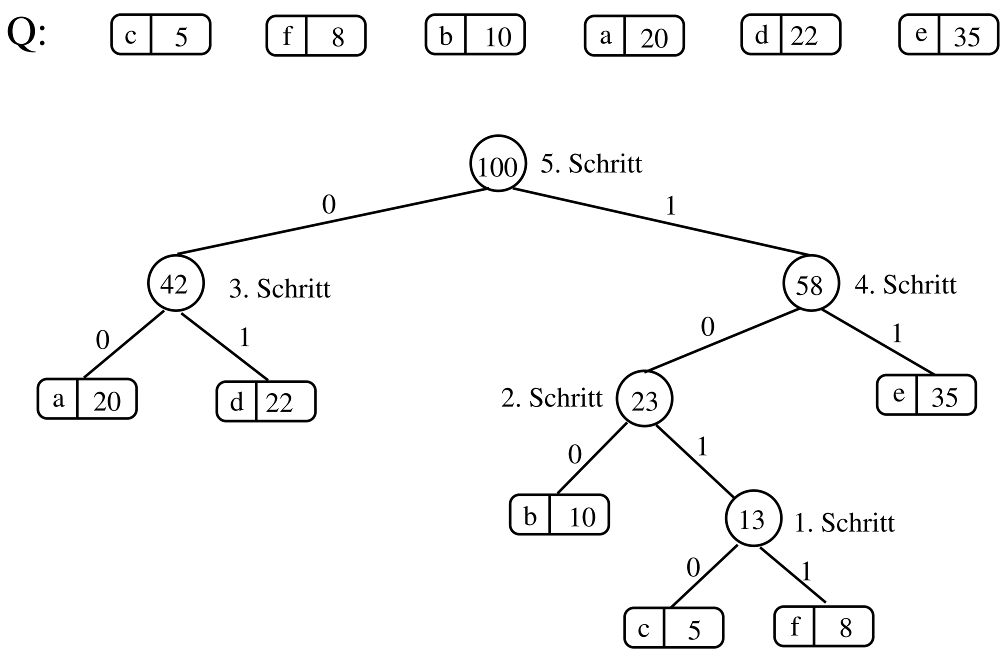
$$
\begin{array}{rll}
T[(\lambda x\, y.x)(\lambda z.y)y] & = \underbrace{T[\lambda x\, y.x]}\ \underbrace{T[\lambda z.y]}\ \underbrace{T[y]} & \text{zweimal (b)} \\
& = abs(x, T[\lambda y.x])\, abs(z, T[y])\, T[y] & \text{zweimal (c)} \\
& = abs(x, T[\lambda y.x])\, abs(z,y)\, y & \text{zweimal (a)} \\
& = abs(x, T[\lambda y.x])\, K\, y\, y & (1) \\
& = abs(x, abs(y, T[x])\, K\, y\, y & (c) \\
& = abs(x, abs(y,x)\, K\, y\, y & (a) \\
& = abs(x, K\, x)\, K\, y\, y & (1) \\
& = S\, abs(x,K) abs(x,x)\, K\, y\, y & (3) \\
& = S(K\, K) I\, K\, y\, y & \text{(2) und (1)}
\end{array}
$$

Aufgabe 5: **Codierungen** $4 + 3$ **Punkte**

a) Bestimmen Sie für die in der Tabelle gegebene Menge $A = \{a, b, c, d, e, f\}$ mit der Häufigkeitsverteilung f den optimalen binären Präfixcode mit dem Huffman-Algoritmus. Skizzieren Sie die einzelnen Schritte und tragen Sie am Ende die Codewörter in die dritte Zeile der Tabelle ein.

Symbol $x \in A$	a	b	c	d	e	f
Häufigkeit $f(x)$	20	10	5	22	35	8
Codierung	00	100	1010	01	11	1011

Lösung: Die folgende Abbildung zeigt oben die geordnete Prioritätswarteschlange Q nach Initialisierung und unten den Codebaum des optimalen Codes. An den inneren Knoten kann die Reihenfolge der Schritte des Huffman-Algorithmus abgelesen werden. Die Codierungen wurden bereits oben in die Tabelle eingetragen.



b) Zeigen Sie dass für jede Menge B mit $n = 2^k$ Elementen $a_1, a_2, \ldots, a_n$ und einer Häufigkeitsverteilung $f(a_1) \leq f(a_2) \leq \ldots \leq f(a_n)$ mit der zusätzlichen Eigenschaft $f(a_n) < 2 \cdot f(a_1)$ der optimale Code ein Blockcode ist (d.h. alle Codewörter haben die gleiche Länge).

Lösung: Beweis mit vollständiger Induktion nach k.

Induktionsanfang: Für $k = 1$ hat B zwei Elemente, die man optimal mit 0 und 1 codiert.

Induktionsschritt: $k \to k + 1$.
Sei $n = 2^{k+1}$ und $B = \{a_1, \ldots, a_n\}$ eine Menge mit der Häufigkeitsverteilung $f(a_1) \leq f(a_2) \leq \ldots \leq f(a_n)$ sowie $f(a_n) < 2f(a_1)$. Um eine optimale Codierung zu erzeu-

gen, führen wir den Huffman-Algorithmus aus und beobachten nach der Initialisierung die ersten $\frac{n}{2}$ Schritte: Auf Grund der Voraussetzungen über f werden erst a_1 und a_2 aus der Warteschlange Q entfernt und unter einer neuen Wurzel a'_1 mit Schlüsselwert $f(a'_1) = f(a_1) + f(a_2)$ in Q eingefügt, dann analog a_3 und a_4 unter neuer Wurzel a'_2 mit $f(a'_2) = f(a_3)+f(a_4)$ und letztlich a_{n-1} und a_n unter $a'_{n/2}$ mit $f(a'_{n/2}) = f(a_{n-1})+f(a_n)$. Wir beobachetn, dass $B' = \{a'_1, a'_2, \ldots, a'_{n/2}\}$ mit $n/2 = 2^k$ die Bedingungen zur Anwendung der Induktionsvoraussetzung erfüllen: Der erste Teil $f(a'_1) \leq f(a'_2) \leq \ldots \leq f(a'_{n/2})$ ist offensichtlich. Darüber hinaus gilt

$$f(a'_{n/2}) = f(a_{n-1}) + f(a_n) \leq 2f(a_1) + 2f(a_1) \leq 2(f(a_1) + f(a_2)) = 2f(a'_1).$$

Folglich erzeugt der Huffman-Algorithmus einen Blockcode für B' und da alle Symbole aus B in Kinderknoten von B'-Knoten stehen, erzeugt der Huffamn-Algorithmus für B auch einen Blockcode.

Aufgabe 6: **Vermischtes** $2+5$ **Punkte**

a) Welche Werte haben die folgenden zwei Haskell-Ausdrücke? Begründungen sind hier nicht erforderlich.

`length[(x,y)|x<-"basic", y<-"test", x>=y]` $\rightsquigarrow$ **Lösung:** `3`
Zusatzkommentar: Liste enthält die Paare `('s','e')`, `('s','s')` und `('i','e')`.

`foldr max 0 [x+y |x<-[1..20],y<-[1,40-x]]` $\rightsquigarrow$ **Lösung:** `40`
Zusatzkommentar: Für jedes x entsteht die größte Summe mit $y = 40 - x$, also ist 40 das (mehrfach auftretende) Maximum in der Liste.

b) Bestimmen Sie den allgemeinst möglichen Typ für die folgenden Ausdrücke und begründen Sie kurz Ihre Antworten:

```
f x y z = "abcd"++ map x [d | d<-y, d<= lentgh z]
```

Lösung:
`f :: (Int->Char)->[Int]->[a]->[Char]` •

Begründung:
Ausgabetyp muss String sein (wegen `"abcd"++...`) und somit muss auch das Ergebnis von `map x [d | d<-y, d<= lentgh z]` ein String sein.
Nach Definition ist `map::(a->b)->[a]->[b]` und somit `x::a->b` , `b=Char` ,
`[d | d<-y, d<= lentgh z]::[a]` und letztlich `y::[a]`. Da die Elemente von `y` mit `length z` vergleichen werden, muss `a=Int` und `z` ein beliebiger Listentyp sein. •⊙

```
g  x y z
   |foldr y (y x x) z  = [ ]
   |otherwise          = [z]
```

Lösung:
`g :: Bool->(Bool->Bool->Bool)->[Bool]->[[Bool]]` •

Begründung: Da `foldr y (y x x) z` als Bedingung auftritt, muss dieser Ausdruck vom Typ `Bool` sein. Nach Definition ist `foldr::(a->b->b)->b->[a]->b` und somit `b=Bool` , sowie `y::a->Bool->Bool` und `x::Bool`. Wegen der Anwendung `y x x` muss `a=b=Bool` sein, also `z::[a]=[Bool]` und da die Ausgabe von `g` im zweiten Fall `[z]` ist, muss der Ausgabetyp von `g` gleich `[[Bool]]` sein. •⊙