Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
3. Juni 2014

7. Musterlösung zur Vorlesung

Programmierung und Modellierung

Hinweise: Am 12.6. um 17h findet in B101 eine Aufholübung statt. Näheres dazu findet sich auf der Vorlesungshomepage.

Die Übungen am 10.6., 11.6. und 20.6.(!) entfallen. Übungen finden statt am:

Übung	**Dienstag**	**Mittwoch**	**Freitag**
7.	3.6.	4.6.	6.6.
8.	17.6.	18.6.	**13.6.**
9.	24.6.	25.6.	27.6.

Bitte verwenden Sie zur Bearbeitung die auf der Vorlesungshomepage bereitgestellten Dateivorlagen. Bearbeiten Sie die mit `-- TODO` markierten Stellen. Sie dürfen auch neue Hilfsfunktion definieren, und alle vorhandenen Funktion nutzen; Sie dürfen aber nicht die importierten Module am Anfang der Datei abändern.

A7-1 *Data.Map* Das Modul `Data.Map` der Standardbibliothek bietet die Verwendung von endlichen Abbildungen über eine Implementierung von balancierten Suchbäumen bereit. In dieser Aufgabe wollen wir lernen, wie wir dieses Modul benutzen können.

Importieren Sie das Modul mit `import qualified Data.Map as Map`, um Namenskonflikte zu vermeiden (siehe dazu auch Folie 6-37). Wichtige Funktionen des Moduls sind:

```
empty  :: Map k a
insert :: Ord k => k -> a -> Map k a -> Map k a
delete :: Ord k => k -> Map k a -> Map k a
lookup :: Ord k => k -> Map k a -> Maybe a
```

weitere nützliche Funktionen finden Sie in der Dokumentation des Untermoduls `Data.Map.Lazy` in Standardbibliothek. Die Unterteilung von `Data.Map` in Untermodule können Sie ignorieren.

a) Dr. Jost möchte zur Erbauung der Moral ein Programm zur Verwaltung der ProMo-Studenten schreiben. Dazu hat er bereits folgendes implementiert:

```
import qualified Data.Map as Map

data Bemerkung = Störer | Abschreiber deriving (Eq, Show)
type Student = String -- zur Vereinfachung, besser newtype/data verwenden!
type Register  = Map.Map Student [Bemerkung]

eintragStörer :: Student -> Register -> Register
eintragStörer stud sreg = undefined

hatGestört :: Student -> Register -> Bool
hatGestört stud sreg = undefined
```

Vervollständigen Sie die Definitionen für `eintragStörer` und `hatGestört`, so dass folgendes funktioniert:

```
> :m + Data.Map
> let reg = eintragStörer "Eustachius" $ eintragStörer "Wally" empty
> hatGestört "Xaver" reg
False
> hatGestört "Wally" reg
True
```

LÖSUNGSVORSCHLAG:

Hier sind einige Lösungsmöglichkeiten:

```
eintragStörer1 :: Student -> Register -> Register
eintragStörer1 s m = case Map.lookup s m of
    Nothing   -> Map.insert s [Störer   ] m
    (Just vs) -> Map.insert s (Störer:vs) m

eintragStörer2 :: Student -> Register ->  Register
eintragStörer2 s = Map.insertWith (++) s [Störer]

hatGestört1 :: Student -> Register -> Bool
hatGestört1 s m = case Map.lookup s m of
    (Just vs) -> Störer `elem` vs
    Nothing   -> False

hatGestört2 :: Student -> Register -> Bool
hatGestört2 s m
  | (Just vs) <- Map.lookup s m = Störer `elem` vs
  | otherwise = False
```

Hinweis: Beachten Sie, dass in einem funktionalen Programm Werte im Speicher nie verändert werden. Dementsprechend bekommt die Funktion `eintragStörer` ein `Register` und liefert immer wieder ein *neues* `Register` zurück:

```
> let reg1 = eintragStörer "Eustachius" $ eintragStörer "Wally" empty
> let reg2 = eintragStörer "Xaver" reg1
> hatGestört "Xaver" reg1
False
> hatGestört "Xaver" reg2
True
```

b) Eustachius hat eine Umkehrfunktion zu `vervielfache` aus Aufgabe A6-2c geschrieben:

```
import Data.List (nub)
count xs = [(x, length [ 1 | x0 <- xs, x0 == x] ) | x <- nub xs]

> count "abcdabcabaa"
[('a',5),('b',3),('c',2),('d',1)]
```

Diese Implementierung ist korrekt, aber leider ineffizient, wie sie leicht testen können:

```
> :set +s
> let testString n = concat (replicate n "Floki fährt furchtlos mit dem
                                          vollbesetzten Flos durch den fulminanten Fjord. ")
> count $ testString 10000
[('F',30000),('l',60000),('o',50000),('k',10000),('i',30000),(' ',110000),('f',30000),('\228',10000),('h'
(4.56 secs, 1149487632 bytes)
```

Reimplementieren Sie `count` durch die Verwendung von `Data.Map` effizienter!

LÖSUNGSVORSCHLAG:

```
count1 :: Ord a => [a] -> [(a,Int)]
count1 xs = Map.assocs $ countAux Map.empty xs
  where
    countAux  :: Ord a => (Map.Map a Int) -> [a] -> (Map.Map a Int)
    countAux m   []  = m
    countAux m (h:t) = case Map.lookup h m of
        Nothing  -> countAux (Map.insert h    1  m) t
        (Just i) -> countAux (Map.insert h (i+1) m) t

count2 :: (Eq a, Ord a) => [a] -> [(a,Int)] -- Pointfree
count2 = Map.assocs . foldl (\m x -> Map.insertWith (+) x 1 m) Map.empty
```

Variante `count1` könnte man auch analog zu `hatGestört2` wieder mit Pattern-Guards implementieren.

A7-2 *Suchbäume* Ein *Suchbaum* ist ein Baum mit Beschriftungen aus einer geordneten Menge, bei dem für jeden Knoten k gilt, dass die Beschriftung von k größer-gleich ist als alle Beschriftungen des linken Teilbaums von k und kleiner-gleich als alle Beschriftungen des rechten Teilbaums. Ein binärer Suchbaum ist balanciert, wenn für jeden Knoten k gilt, sich die Höhe der beiden Teilbäume von k höchstens um 1 unterscheidet.

Die Funktion `isBalanced` entscheidet, ob ein Binärbaum balanciert ist (die Suchbaumeigenschaft selbst wird hier zur Vereinfachung nicht geprüft):

```
data Tree a = Empty | Node { label :: a, left,right :: Tree a }
deriving (Show, Eq)

tiefe :: Tree a -> Int
```

```
tiefe Empty = 0
tiefe Node {left=l,right=r} = 1 + max (tiefe l) (tiefe r)

isBalanced :: Tree a -> Bool
isBalanced Empty = True
isBalanced Node {left=l,right=r} =
  let hdiff = (tiefe l) - (tiefe r)
  in isBalanced l && isBalanced r && (abs hdiff <= 1)
```

Die Funktion `abs : int -> int` liefert den absoluten Betrag des Arguments, also gilt z.B. `abs 5 = 5` als auch `abs (-5) = 5`.

a) Geben Sie die worst-case-Laufzeitkomplexität von `isBalanced` an.

LÖSUNGSVORSCHLAG:

Betrachten wir zuerst `tiefe`, da diese aufgerufen wird. Die Funktion `tiefe` besucht jeden Knoten des Baumes, hat also eine Komplexität von $O(n)$.

Der worst-case bei `isBalanced` wird für einen balancierten Baum erreicht – ansonsten terminiert die Funktion schon vorher, wenn der Baum als unbalanciert erkannt wird. Für einen balancierten Baum wird jeder Knoten betrachtet (d.h. ein Aufruf von `isBalanced` für jeden Knoten), und für jeden Knoten werden bei den Aufrufen von `tiefe` die Knoten seiner jeweiligen Teilbäume besucht.

Dies ergibt aber nicht $O(n^2)$, da der Algorithmus ja nicht für jeden Knoten alle anderen Knoten betrachtet, sondern nur die Knoten seiner Teilbäume. Wie wir eingangs beobachtet haben, findet der worst-case bei einem balancierten Baum statt. Dies bedeutet, dass sich die Größe der Eingabe von `tiefe` sich jedem Schritt ungefähr halbiert, also logarithmisch in n ist. Da `tiefe` selbst eine lineare Laufzeitkomplexität besitzt, ist die Laufzeit bei eine Eingabe der Größe $\log n$ unverändert $\log n$. Daraus ergibt sich somit eine gesamte Laufzeitkomplexität von $O(n \cdot \log n)$ für `isBalanced`.

b) Geben Sie eine alternative Definition von `isBalanced` an, welche lineare Zeitkomplexität besitzt, d.h. jeden Knoten nur einmal betrachtet. Z.B. betrachtet die Funktion `foldTree` jeden Knoten eines Baumes nur einmal. (Die Suchbaumeigenschaft muss weiterhin nicht geprüft werden, sondern nur die Eigenschaft, balanciert zu sein.)

LÖSUNGSVORSCHLAG:

Die Idee hierbei ist, dass man die Information über die maximale Tiefe der Teilbäume mit nach oben reichen kann. Wir schreiben hierzu also eine Hilfsfunktion, die sowohl die maximale Tiefe zurückgibt, als auch ein Flag, das speichert, ob der Check im Teilbaum bisher erfolgreich verlief: Hier wird jeder Knoten nur noch einmal besucht – und man erhält insgesamt lineare Komplexität, $O(n)$, obwohl die Funktion nicht endrekursiv ist.

```
isBalanced :: Tree a -> Bool
isBalanced = snd . foldTree isBalancedaux (0,True)
  where
    isBalancedaux ::((Int,Bool), a, (Int,Bool)) -> (Int,Bool)
    isBalancedaux ((hl,avll),_,(hr,avlr)) =
      let height = 1 + max hl hr
          diff   = abs $ hl - hr
      in (height, avll && avlr && diff <= 1)
```

Natürlich muss man `foldTree` nicht unbeding einsetzen:

```
isBalanced2 :: Tree a -> Bool
isBalanced2 = snd . isBalancedaux
  where
    isBalancedaux :: Tree a -> (Int, Bool)
    isBalancedaux Empty = (0,True)
    isBalancedaux Node {left=l,right=r} =
        let (hl,avll) = isBalancedaux l
            (hr,avlr) = isBalancedaux r
            height    = 1 + max hl hr
            diff      = abs $ hl - hr -- 2 * max - hl - hr =
        in (height, avll && avlr && diff <= 1)
```

A7-3 ***Typregel*** Überlegen Sie sich analog zu den in der Vorlesung behandelten Typregeln eine sinnvolle Typregel für den Haskell Ausdruck `let ... = ... in ...`

a) Die Konklusion der Typregel ist bereits vollständig vorgegeben. Ergänzen Sie lediglich die fehlende(n) Prämisse(n):

$$\frac{}{\Gamma \vdash \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2\ ::\ C} \qquad \text{(Let)}$$

Hinweis: Aus der vorgegeben Konklusion der Typregel folgt auch, dass Ihre Typregel kein volles Pattern-Matching wie in Haskell erlauben muss. Es reicht uns hier zur Vereinfachung, wenn der Unterausdruck e_1 mit einer Variablen x gematched wird.

LÖSUNGSVORSCHLAG:

Da der gesamte Ausdruck zu e_2 auswertet, hat dieser den gleichen Typ wie e_2. Wir müssen also fordern, dass e_2 ebenfalls den Typ C hat.

Allerdings darf e_2 die neu eingeführte Variable x verwenden, d.h. wir müssen den Kontext für e_2 um x erweitern. Dabei wissen wir nicht, welchen Typ x hat; aber wir wissen,

dass dies der gleiche Typ wie von e_1 sein muss, denn x wertet ja zu e_1 aus. Dementsprechend benötigen wir eine zweite Prämisse für den Typ von e_1.

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma,\, x{::}A \vdash e_2 :: C}{\Gamma \;\vdash\; \texttt{let}\; x \;=\; e_1 \;\texttt{in}\; e_2 \;::\; C} \qquad \text{(Let)}$$

b) Beachten Sie, dass Haskell rekursive let-Definitionen wie z.B.
`let ones = 1 : ones in take 111 ones` erlaubt. Ihre Typregel soll dies ebenfalls erlauben, ändern Sie Ihre Typregel aus der vorherigen Teilaufgabe entsprechend ab, falls Typregel dies noch nicht erlaubt.

LÖSUNGSVORSCHLAG:

Für die rekursive Variante bekommt die Typherleitung von e_1 noch das Typurteil $x :: A$ hinzu, da die Definition von `x` ja auf sich selbst Bezug nehmen darf.

$$\frac{\Gamma,\, x{::}A \vdash e_1 :: A \qquad \Gamma,\, x{::}A \vdash e_2 :: C}{\Gamma \;\vdash\; \texttt{let}\; x \;=\; e_1 \;\texttt{in}\; e_2 \;::\; C} \qquad \text{(Let)}$$

H7-1 *Data.Map II* (3 Punkte) (`.hs`-Datei als Lösung abgeben)
Fortsetzung von Aufgabe A7-1. Sie dürfen alle Funktion des Moduls `Data.Map` und dessen Untermodule nutzen, so wie alle Funktionen der `Prelude`, wenn Sie möchten. Achten Sie auf eine effiziente Implementierung!

a) Implementieren Sie die Funktion `eintragAbschreiber :: [Student] -> Register -> Register` welche einer Liste von Studenten den Eintrag `Abschreiber` in das Register hinzufügt.

Beispiel:

```
> eintragAbschreiber ["Eustachius","Ignaz"] $
  eintragAbschreiber ["Ignaz","Edeltraud"]  $ Map.empty
fromList [("Edeltraud",[Abschreiber]),
          ("Eustachius",[Abschreiber]),
          ("Ignaz",[Abschreiber,Abschreiber])]
```

LÖSUNGSVORSCHLAG:

```
eintragAbschreiber1 :: [Student] -> Register ->  Register
eintragAbschreiber1   []  reg = reg
eintragAbschreiber1 (h:t) reg =
  case Map.lookup h reg of
    Nothing   -> eintragAbschreiber1 t $ Map.insert h [Abschreiber   ] reg
    (Just vs) -> eintragAbschreiber1 t $ Map.insert h (Abschreiber:vs) reg

eintragAbschreiber2 :: [Student] -> Register ->  Register
eintragAbschreiber2 studs reg = foldl aux reg studs
  where aux :: Register -> Student -> Register
        aux r s = Map.insertWith (++) s [Abschreiber] r
```

b) Implementieren Sie die Funktion `gesamtRegister :: [Register] -> Register` welche die Eintragungen der Register mehrerer Vorlesungen zu einem einzigen Register kombiniert.

LÖSUNGSVORSCHLAG:

```
gesamtRegister :: [Register] -> Register
gesamtRegister = Map.unionsWith (++)
```

H7-2 *Suchbäume II* (2 Punkte) (`.hs`-Datei als Lösung abgeben)
Ein *AVL-Baum* ist ein binärer Suchbaum, in dem für jeden Knoten k gilt, dass sich die Höhe der beiden Teilbäume von k höchstens um 1 unterscheidet. Um die Balance effizient zu halten, trägt jeder Knoten zusätzlich eine Zahl, welche die Höhendifferenz seiner beiden Unterbäume angibt Wir vereinbaren dabei die Konvention: ein positiver Wert steht dafür, dass der rechte Unterbaum größer ist; ein negativer Wert steht dafür, dass der linke Unterbaum größer ist.

```
data AVL a = Empty | Node { label :: a, left,right :: AVL a, balance :: Int }
```

AVL-Bäume wurden 1962 von **A**delson-**V**elsky und **L**andis als erste Datenstruktur für balancierte Suchbäume vorgeschlagen.

Schreiben Sie eine effiziente Funktion `isAVL :: AVL a -> Bool` welche entscheidet, ob ein Binärbaum ein AVL-Baum ist, d.h. wie in Aufgabe A7-2 muss geprüft werden, ob der Baum ausreichend balanciert ist. Zusätzlich ist jedoch zu prüfen, ob die Balance Annotation korrekt ist. Sie müssen jedoch nicht überprüfen, ob die Knoten-Label der Suchbaumeigenschaft genügen.

LÖSUNGSVORSCHLAG:

Die Lösung ist nahezu identisch zu A7-2, wir müssen lediglich noch die gespeicherte Balance mit der bereits ausgerechneten Differenz vergleichen:

```
isAVL :: AVL a -> Bool
isAVL = snd . isAVLaux
  where
    isAVLaux :: AVL a -> (Int, Bool)
    isAVLaux Empty = (0,True)
    isAVLaux Node {left=l,right=r,balance=b} =
        let (hl,avll) = isAVLaux l
            (hr,avlr) = isAVLaux r
            height    = 1 + max hl hr
            diff      = hr - hl
        in (height, avll && avlr && b==diff && (abs diff) <= 1)
```

H7-3 *Substitution* (3 Punkte) (`.hs`-Datei als Lösung abgeben)
Gegeben ist folgende Datentypdeklaration zur Repräsentation von Termen:

```
data Term = Var Char | Const Int | App Term Term | Abs Char Term
```

a) Schreiben Sie eine Funktion `freeVars :: Term -> [Char]` welche die freien Variablen eines Terms effizient berechnet. *Hinweis:* Die Aufgabe ist sehr einfach, wenn Sie den Unterschied zwischen freien und gebundenen Variablen verstanden haben, siehe dazu auch Folien 10-9 und 10-18.

LÖSUNGSVORSCHLAG:

```
freeVars :: Term -> [Char]
freeVars (Var x)     = [x]
freeVars (Const _)   = []
freeVars (App e1 e2) = (freeVars e1) ++ (freeVars e2)
freeVars (Abs x  e1) = filter (/=x) (freeVars e1)
```

b) Implementieren Sie die Substitution von Folie 10-8 effizient als Funktion `subst :: (Char, Term) -> Term -> Term`. Dabei steht `subst ('x', t1) t2` für den Term, den man erhält, wenn alle freien Vorkommen von `Var 'x'` in `t2` durch `t1` ersetzt werden.

Hinweise: Bitte beachten Sie, dass Folie 10-8 leider einen Tippfehler enthielt, welcher ab 3.6.14 korrigiert wurde. Laden Sie die Folien ggf. frisch von der Vorlesungshomepage herunter.

Verwenden Sie zur Vereinfachung folgende partielle Funktion zur Erzeugung frischer Variablen:

```
genFreshV :: [Char] -> Char
genFreshV vs = head $ filter (\c -> not $ c `elem` vs) ['a'..'z']
```

LÖSUNGSVORSCHLAG:

Der einzige schwierige Fall ist die Abstraktion, bei der drei Fälle zu unterscheiden sind:

i) Ist die zu ersetzende Variable identisch mit der durch die Abstraktion gebundene Variable, so ist nichts zu tun, da der Funktionsrumpf ja nur die frisch gebundene Variable referenzieren kann. Substitution ersetzt ja nur frei vorkommende Variablen, den nur freie Variablen referenzieren Dinge außerhalb des betrachteten Bereiches.

ii) Der Term, der in der Substitution eine Variable ersetzt, kann freie Variablen enthalten. Diese Referenzen nach "außerhalb" sollen unverändert erhalten bleiben, d.h. wir müssen aufpassen, dass diese freien Variablen frei bleiben und nicht versehentlich "eingefangen" werden.

Dieser Fall tritt ein, falls die durch die Abstraktion gebundene Variable in dem substituierenden Term frei vorkommt. Um den Konflikt aufzulösen, ändern wir einfach den Namen der gebundenen Variable durch einen frischen - wiederum mithilfe einer Substitution. Danach führen wir noch die eigentliche Substitution auf dem Funktionsrumpf aus.

Grundlegend ist dabei, dass die Umbenennung gebundener Variablen unbedeutend ist: `\x -> e` ist ja äquivalent zu `\y -> e`[y/x].

iii) Bestehen keine Namenskonflikte, dann setzen wir die Substitution wie gehabt auf dem Funktionsrumpf fort.

```
subst :: (Char, Term) -> Term -> Term
subst (x,e) o@(Var y)
  | x == y    = e
  | otherwise = o
subst (x,e) (Const c) = Const c
subst s o@(App e1 e2) = App (subst s e1) (subst s e2)
subst s@(x,e) o@(Abs y e1)
  | x == y        = o
  | y `elem` fv_e = Abs freshV (subst (x,e) $ subst (y, Var freshV) e1)
  | otherwise     = Abs y (subst s e1)
  where
    fv_e   = freeVars e
    fv_e1  = freeVars e1
    freshV = genFreshV (fv_e ++ fv_e1)
```

```
                                    (Char,Term)
[(Char,Term)] Char -> Term      Data.Map.Map Char Term
```

Abgabe: Lösungen zu den Hausaufgaben können bis Donnerstag, den 12.06.2014, 11:00 Uhr mit UniworX abgegeben werden.

Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.