Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
30. Juni 2015

10. Musterlösung zur Vorlesung

Programmierung und Modellierung

Hinweise:

- Einmalige Raumänderung für die Übungen am Freitag, den 3.7.2015: 10:00h→E216, 12:00h→E216, 14:00h→A014 im gleichen Gebäude.
- Donnerstag, 9.7., 17:00, B201: Freiwillige Fragestunde der ProMo-Tutoren!

A10-1 ***Wiederholung*** Rechnen Sie zügig mit Papier & Bleistift aus, zu welchem Wert der jeweilige Haskell Programmausdruck auswertet. Es reicht die Angabe des Endergebnis. Sie dürfen Ihnen unbekannte Funktionen dabei gerne in der Standardbibliothek nachschlagen!

a)
```
let bar x y z = if 1 < succ x then z else x+y
in  bar 0 2 (3 `div` 0)
```

LÖSUNGSVORSCHLAG:

```
2
```

Gemäß unserem Substitutionsmodell, Folien 03-3ff., dürfen wir uns den Teilausdruck zur Auswertung frei auswählen; also z.B. könnte man auch zuerst versuchen, `3 `div` 0` auszuwerten. In Haskell wird aber nicht zufällig ein Teilausdruck zum auswerten ausgewählt, sondern nach einer speziellen Reihenfolge vorgegangen, wie wir in Kapitel 11 sehen werden.

b)
```
[(x,y,z)| x<-[minBound..maxBound], y<-[7,8], z<-[2,3], x==(even(y+z))]
```

LÖSUNGSVORSCHLAG:

```
[(False,7,2),(False,8,3),(True,7,3),(True,8,2)]
```

c)
```
(\x y -> reverse $ ():x:():[()]) (()) ()
```

LÖSUNGSVORSCHLAG:

```
[(),(),(),()]
```

d) `let foo f = \x-> x : foo f (f x) in take 3 $ foo (*2) 2`

LÖSUNGSVORSCHLAG:

`[2,4,8]`

e)
```
let m = Data.Map.fromList [('a',4),('b',7),('c',6),('d',9)]
in Control.Monad.mapM_ (Prelude.flip Data.Map.lookup m) ['b'..'c']
```

LÖSUNGSVORSCHLAG:

`Just ()`

Hier arbeitet die `Maybe`-Monade. Das Ergebnis ist also entweder `Nothing` oder `Just`-irgendwas. Welcher fall eintritt hängt nur davon ab, ob mindestens einmal zwischendruch `Nothing` auftritt.

`mapM` und `mapM_` wenden eine monadische Funktion auf eine Liste an; im gegebenen Code bedeutet dies, dass `'b'` und `'c'` in der Liste nachgeschlagen werden. Da diese beiden Schlüssel in `m` enthalten sind, kommt also schon mal `Just`-irgendwas heraus.

Bei `mapM` würde die Liste der Werte für die beiden Schlüssel herauskommen, d.h. `Just [7,6]`. Die Funktion `mapM_` verwirft jedoch das funktionale Ergebnis und liefert nur `()` zurück, eingepackt im Kontext der Monade, d.h. `Just` oder `Nothing` bleibt unverändert erhalten. Es kommt also `Just ()` heraus.

Die `in`-expression testet also lediglich, ob `['b'..'c']` in der Map `m` enthalten sind.

A10-2 ***Fizz buzz*** Im Kinderspiel "Fizz buzz" sitzen alle Teilnehmer in einem Kreis; ein Spieler beginnt und sagt "1", der nächste Spieler sagt dann schnell die nächsthöhere Zahl. Falls die Zahl jedoch durch 3 teilbar ist, so muss der Spieler "fizz" sagen. Falls die Zahl durch 5 teilbar ist, so muss der Spieler "buzz" sagen. Ist die Zahl sowohl durch 3 als auch durch 5 teilbar, so muss "fizz buzz" gesagt werden. Wer einen Fehler macht, scheidet aus!

Schreiben Sie fix ein Haskell Programm, welches dieses Spiel für die Zahlen 1 bis 111 ausführt. Dabei wird in jede Antwort in einer eignen Zeile wiedergegeben:

```
1
2
fizz
4
buzz
fizz
7
```

Versuchen Sie eine Version dieses Programmes zu erstellen, welche möglichst kurz und ohne direkte rekursive Aufrufe auskommt! Verwenden Sie also die in der Vorlesung behandelten Funktionen aus Modul `Control.Monad`

LÖSUNGSVORSCHLAG:

Viele Lösungen sind hier möglich:

```
main1 = forM_ [1..111] $ \x -> do
  let xm3 = x `mod` 3 == 0
      xm5 = x `mod` 5 == 0
  when xm3 $ putStr "fizz "
  when xm5 $ putStr "buzz "
  if (xm3 || xm5) then putStrLn " " else print x

main2 = forM [1..111] $ putStrLn . fb
  where fb x | xm3 && xm5 = "fizz buzz"
             | xm3        = "fizz"
             |        xm5 = "buzz"
             | otherwise  = show x
            where
              xm3 = x `mod` 3 == 0
              xm5 = x `mod` 5 == 0

main3 :: IO ()
main3 = mapM_ putStrLn list
  where list  = map fun [1..111]
        fun x = case (x `mod` 3, x `mod` 5) of
                  (0,0) -> "fizz buzz"
                  (0,_) -> "fizz"
                  (_,0) -> "buzz"
                  _     -> show x
```

A10-3 ***Fehler-Monade*** Machen Sie den folgenden Datentyp `Entweder` zur Monade:

```
import Control.Applicative
import Control.Monad

data Entweder a b = Eines b | Anderes a deriving (Show, Eq, Ord)
```

Die Grundidee dieser Monade ist wie bei `Maybe`: eine erfolgreiche Berechnung liefert einen mit `Eines` verpackten Wert, währned ein Fehler durch die Rückgabe von `Anderes` signalisiert wird. Während die `Maybe`-Monade bei einem Fehler nur `Nothing` zurückliefert, könnte hier `Anderes` noch eine Fehlerbeschreibung zusätzlich liefern. Versuchen Sie zur Lösung dieser Aufgabe möglichst wenig in Vorlesungskapitel 09 nachzuschlagen!

Beispiel:

```
> (*) <$> (Eines 3) <*> (Eines 4)
Eines 12
> let foo x y = if y>0 then Eines $ x `div` y else Anderes "Div-by-Zero"
> foldM foo 100 [2,5,3]
Eines 3
> foldM foo 100 [2,5,0,3]
Anderes "Div-by-Zero"
```

Hinweise:

- Welchen Kind hat der Typkonstruktor `Entweder`? Welchen Kind benötigt die Instanzdeklaration für die Monade?

LÖSUNGSVORSCHLAG:

Der Kind ist `Entweder :: * -> * -> *`, aber die Monade benötigt Kind `* -> *`, d.h. in der Instanzdeklaration müssen wir das erste Typargument durch eine Typvariable füllen: `instance Monad (Entweder a) where`

- Berücksichtigen Sie die Monaden-Gesetze!
- Der Wert `Anderes "foo"` des Typs `Entweder String Int` kann nicht einfach als Wert des Typs `Entweder String Double` aufgefasst werden! Hier muss umverpackt werden, d.h. den Konstruktor `Anderes` erst entfernen, danach wieder erneut davor setzen. Je nach Typ wird ja auch eine andere Menge an Speicherplatz reserviert. Fehlermeldungen wie `Couldn't match type 'a1' with 'b'...` oder `Could not deduce (b ~ a1)...` weisen auf dieses Problem hin.

LÖSUNGSVORSCHLAG:

Dieser Typ ist in der Standardbibliothek als `Either` bekannt und dort ganz genauso als Monade deklariert.

Da die `Monad`-Instanzdeklaration eine Deklaration für `Applicative` voraussetzt und dieser weider eine `Functor`-Instanz fordert, gibt es zwei Arten, diese Aufgabe zu lösen, je nachdem ob man untern oder oben oder in der Klassenhierarchie beginnt. Hier die konzeptionell bessere Methode von unten nach oben:

```
instance Functor (Entweder a) where
  fmap f (Eines b)   = Eines $ f b
  fmap _ (Anderes a) = Anderes a   -- Umpacken notwendig!

instance Applicative (Entweder a) where
  pure = Eines
  (Eines f)   <*> r = fmap f r
  (Anderes a) <*> _ = Anderes a    -- Umpacken notwendig!

instance Monad (Entweder a) where
  return = pure
  (Anderes a) >>= _ = Anderes a    -- Umpacken notwendig!
  (Eines b)   >>= k = k b
```

Hier nun die traditionelle Methode, bei der die (applikativen) Funktoren einfach aus der mächtigen Bind-Operation der Monade erzeugt werden (bei GHC < 7.10 kann man die letzten beiden Instanzdeklarationen weglassen):

```
instance Monad (Entweder a) where
  return = Eines

  (Eines   x) >>= m = m x
  (Anderes y) >>= _ = Anderes y    -- Umpacken notwendig!

instance Applicative (Entweder a) where
  pure  = return
  (<*>) = ap

instance Functor (Entweder a) where
  fmap = liftM
```

H10-1 ***AVL Bäume*** (0 Punkte; Datei `H10-1.hs` als Lösung abgeben)
In der Vorlesung am 29.6.2015 wurden verschiedene Optimierungen von balancierten Suchbäumen besprochen. Implementieren Sie gemäß der Beschreibung in Kapitel 07 eine Funktion zum Einfügen eines Wertes in einen AVL-Baum

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

avl_insert :: Ord a => a -> AVL a -> AVL a
```

Hinweis: Diese Aufgabe mag etwas schwerer sein, ist aber durchaus auch lehrreich. Wer nicht weiter kommt findet eine Dateivorlage auf der Vorlesungshomepage, bei der schon viel vorgegeben wurde.

LÖSUNGSVORSCHLAG:

Vorsicht: Wer Lösungen zu dieser Aufgabe im Web sucht, wird viele Lösungen finden, welche beim Einfügen andauern die Höhe/Tiefe der Teilbäume neu berechnen. Dies ist natürlich unsinnig und führt zu einer wesentlichen Verschlechterung der Laufzeit!

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

leaf :: a -> Tree a
leaf a = Node {label=a, left=Empty, right=Empty, balance=0}

ins_BST :: Ord a => a -> Tree a -> Tree a
ins_BST x t = fst $ ins_aux  x t
  where
    ins_aux :: Ord a => a -> Tree a -> (Tree a, Int)
    ins_aux e Empty = (leaf e, 1)
    ins_aux e n@(Node nx tl tr b)
      | e == nx = (n,0)
      | e <  nx = -- BST: links einfügen
        let (l1,hdiff) = ins_aux e tl
        in case Node nx l1 tr (b-hdiff) of                          -- ACHTUNG: (b-hdiff) nicht mehr in {-1,0,1}
              -- Hinweis: Damit die Bennenung zu den Folien 07-51ff passt,
              --          bauen wir hier zuerst den Knoten zusammen und matchen diesen gleich wieder.
              n1@(Node _ _ _ b1) | -2<b1, b1<2 ->               -- Balance ist akzeptabel
                  (n1,if hdiff==1 && b == 0 then 1 else  0)
              Node x (Node y r s 0) t (-2) ->                   -- Fall 1,  Folie 07-51
                  (Node  y r (Node x s t (-1)) 1, 1)
              Node x (Node y r s (-1)) t (-2) ->                -- Fall 2,  Folie 07-52
                  (Node  y r (Node x s t 0) 0, 0)
              Node x (Node y r (Node z u v 0) 1) t (-2) ->      -- Fall 3a, Folie 07-53
                  (Node z (Node y r u 0) (Node x v t 0) 0, 0)
              Node x (Node y r (Node z u v 1) 1) t (-2) ->      -- Fall 3b, Folie 07-53
                  (Node z (Node y r u (-1)) (Node x v t 0) 0, 0)
              Node x (Node y r (Node z u v (-1)) 1) t (-2) ->   -- Fall 3c, Folie 07-53
                  (Node z (Node y r u 0) (Node x v t 1) 0, 0)
      | e >  nx = -- BST: rechts einfügen
        let (r1,hdiff) = ins_aux e tr
        in case Node nx tl r1 (b+hdiff) of                          -- ACHTUNG: (b-hdiff) nicht mehr in {-1,0,1}
              n1@(Node _ _ _ b1) | -2<b1, b1<2 ->               -- Balance ist okay
                  (n1,if hdiff==1 && b == 0 then 1 else  0)
              Node x t (Node y s r 0) 2 ->                      -- Fall 1,  Folie 07-51
                  (Node y (Node x t s (-1)) r (-1), 1)
              Node x t (Node y s r 1) 2 ->                      -- Fall 2,  Folie 07-52
                  (Node y (Node x t s 0) r 0, 0)
              Node x t (Node y (Node z v u 0) r (-1)) 2 ->      -- Fall 3a, Folie 07-53
                  (Node z (Node x t v 0) (Node y u v 0) 0, 0)
              Node x t (Node y (Node z v u 1) r (-1)) 2 ->      -- Fall 3b, Folie 07-53
                  (Node z (Node x t v (-1)) (Node y u r 0) 0, 0)
              Node x t (Node y (Node z v u (-1)) r (-1)) 2 ->   -- Fall 3c, Folie 07-53
                  (Node z (Node x t v 0) (Node y u r 1) 0, 0)
```

Hinweis: AVL-Bäume sind in imperativen Sprachen, welche einen update der Balance-Werte erlauben, durchaus sinnvoll. Für eine rein funktionalen Sprache wie Haskell sind die Updates natürlich nicht so toll; hier empfehlen sich Alternativen, wie z.B. persistente Splay Trees.

H10-2 ***Monadische Komposition*** (4 Punkte; Datei `H10-2.hs` als Lösung abgeben)

a) Schreiben Sie eine monadische Funktion `frage :: String -> IO String`, welche Ihr Argument auf den Bildschirm ausgibt und gleich danach den Benutzer nach einer Antwort fragt, und diese als Ergebnis zurück liefert. Verwenden Sie die DO-Notation!

LÖSUNGSVORSCHLAG:

```
import System.IO

frage1 question = do  -- Einfachste Antwort
  putStrLn question
  getLine

frage2 question = do  -- hFlush Beispiel
  putStr question
  putChar ' '
  hFlush stdout
  getLine
```

b) Schreiben Sie folgende Infix-Funktion aus Modul `Control.Monad` selbst:

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
```

Diese Funktion erwartet zwei monadische Funktionen als Argument und verschmilzt diese zu einer einzigen. *Vorlage:*

```
-- import Control.Monad -- VERBOTEN
infixr 1 <=<
_ <=< _ = undefined -- TODO
```

Hinweise:

- Diese Teilaufgabe ist dank DO-Notation auch nicht schwerer als die Teilaufgabe a)!
- In **allen Teilaufgaben dieser Aufgabe** sind keinerlei Imports erlaubt! Die gesuchten Funktionen sind ja in Modul `Control.Monad` bereits enthalten, so dass nichts mehr zu tun wäre. (Für die späteren Teilaufgaben können `(>>=)` und `return` verwendet werden, da diese bereits durch Modul `Prelude` zur Verfügung stehen.)
- Verwenden Sie `infixr 1 <=<` zur Deklaration der Präzedenz des Infix-Operators.

LÖSUNGSVORSCHLAG:

```
y <=< x = \a -> do b <- x a
                   y b
```

c) Schreiben Sie eine Funktion `frageV2 :: String -> IO String`, welche funktioniert wie in Teilaufgabe a), aber vewenden Sie dieses Mal anstatt der DO-Notation Ihre neu definierte Infix-Funktion `(<=<)`!

Hinweise: Dabei gibt es das Problem, dass die Funktion `getLine` kein Argument erwartet, und daher vom Typ nicht als Argument für `(<=<)` geeignet ist. Dies kann man jedoch leicht mit einer anonymen Funktion lösen! Diese Aufgabe dient zur Übung, die Verwendung von `(<=<)` ist in diesem Fall vielleicht etwas unsinnig.

LÖSUNGSVORSCHLAG:

```
frageV2 = (\_-> getLine) <=< putStrLn
```

Anstelle von `(\_ -> getLine)` könnten wir auch einfacher `const getLine` schreiben.

d) Lösen Sie Teilaufgabe b) erneut, ohne dabei die DO-Notation zu verwenden. Benutzen Sie stattdessen den Bind-Infix-Operator `(>>=)`. Benennen Sie zur Unterscheidung diese zweite Version `komp2 :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)`

LÖSUNGSVORSCHLAG:

```
komp2 f g = (\x -> g x >>= f)
```

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 7.07.2015, 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.