Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
19. Mai 2015

5. Musterlösung zur Vorlesung

Programmierung und Modellierung

Hinweise:

- Ab sofort dürfen alle Funktionen des Moduls `Prelude` verwendet werden, so fern in einer Aufgabe nichts anderes angegeben wurde.
- Die Bearbeitungszeit der Hausübungen für dieses Blatt beträgt 14 Tage.
- Übungsblatt 6 wird aufgrund der Feiertage am Do 28.5., Fr 29.5., Di 2.6. und Mi 3.6. behandelt werden. Es entfallen außerplanmäßig die Übungen am Mi 27.5. und Fr 5.6.

A5-1 *Listenverarbeitung höherer Ordnung*

a) Ersetzen Sie die List-Comprehension in der folgenden Definition durch den Einsatz der Funktionen `map` und `filter` aus der Standardbibliothek:

```
foo1 f p xs = [f x | x <- xs, x >= 0, p x]
```

LÖSUNGSVORSCHLAG:

```
foo2 f p xs = map f (filter p (filter (>=0) xs))

foo3 f p xs = map f (filter (\x-> p x && x >=0) xs)
```

Hinweis: Alle drei Versionen sind nahezu gleich effizient mit GHC, wenn alle Optimierungen eingeschaltet sind (Stichwort: “List-Fusion”).

b) Die Funktion `dropWhile :: (a -> Bool) -> [a] -> [a]` aus der Standardbibliothek entfernt so lange Element vom Anfang einer Liste, so lange diese das gegebene Prädikat erfüllen. Implementieren Sie diese Funktion selbst mit Rekursion, ohne die Verwendung von Bibliotheksfunktionen.

Beispiel: `dropWhile (<4) [1,3,4,5,3,1] == [4,5,3,1]`

LÖSUNGSVORSCHLAG:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] =  []
dropWhile p xs@(x:xs')
  | p x        =  dropWhile p xs'
  | otherwise  =  xs
```

Wer `@`-Patterns nicht kann, schaut entweder auf Folie 02-29 nach oder schreibt alternativ:

```
dropWhile2 :: (a -> Bool) -> [a] -> [a]
dropWhile2 p (x:xs) | p x = dropWhile2 p xs
dropWhile2 _ l = l

dropWhile3 :: (a -> Bool) -> [a] -> [a]
dropWhile3 p = dW3aux
  where dW3Aux (x:xs) | p x = dW3Aux xs
        dW3Aux    xs        = xs
```

c) Die Funktion `all :: (a -> Bool) -> [a] -> Bool` aus der Standardbibliothek gibt nur dann `True` zurück, wenn alle Element der Liste das übergebene Prädikat erfüllen. Implementieren Sie die Funktion ohne direkte Rekursion, sondern unter Verwendung Funktionen höherer Ordnung aus der Standardbibliothek.

LÖSUNGSVORSCHLAG:

```
all1 p xs = foldr aux True xs
  where
    aux x acc
      | p x       = acc
      | otherwise = False

-- Etwas punktfreier ohne xs geht es hier auch:
all2 p = foldr aux True
  where
    aux x acc
      | p x       = acc
      | otherwise = False
```

```
-- Mit anonymer Funktion geht es etwas kompakter:
all3 p = foldr (\x acc -> p x && acc) True

-- Wer sich in der Standardbibliothek gut auskennt,
-- schreibt ganz Punkt-frei mit Punkt-Operator einfach:
all p = and . map p
```

Man könnte auch analog das endrekursive `foldl` verwenden. Allerdings ist `foldr` hier oft effizienter, da die Berechnung ja sofort abbricht, so bald ein Listenelement das Prädikat `p` nicht erfüllt. Für `foldl` gilt das zwar prinzipiell auch, aber `foldl` beginnt am Ende der Liste und muss diese dadurch trotzdem komplett durchlaufen.

A5-2 *Funktionen höherer Ordnung*

a) Implementieren Sie folgende Funktionen:

```
curry3   :: ((a, b, c) -> d) -> a -> b -> c -> d
uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
```

Hinweis: Die Typsignaturen lassen bei der Implementation einer totalen Funktion hier schon keine Wahl mehr zu, wenn man auf Schummeleien wie `undefined`, `error` oder endlose Rekursion verzichtet.

LÖSUNGSVORSCHLAG:

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry3   f  a b c  = f (a,b,c)

uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (a,b,c) = f  a b c
```

b) Diskutieren Sie anhand des Typs den Unterschied zwischen folgenden drei Funktionen:

i) `uncurry3 foldr`

ii) `uncurry (uncurry foldr)`

iii) `(uncurry . uncurry) foldr`

LÖSUNGSVORSCHLAG:

```
uncurry3 foldr            :: ( a -> c -> c, c , [a]) -> c
uncurry (uncurry foldr)   :: ((a -> c -> c, c), [a]) -> c
(uncurry . uncurry) foldr :: ((a -> c -> c, c), [a]) -> c
```

b)i und b)ii unterscheiden sich lediglich in der Gruppierung der Argumente: einmal haben wir die Struktur `(a,b,c)` und einmal `((a,b),c)`. In der Mathematik unterscheidet man oft nicht zwischen diesen beiden Produkten. In einer Programmiersprache kann man eine Unterscheidung aber nur schwerlich vermeiden, schon allein wegen der Unterschiedlichen Kodierung im Speicher des Computers, welche je nach Verwendung notwendig ist.

Die Funktionen b)ii und b)iii sind dagegen tatsächlich identisch. Es ist ja generell egal, ob man zuerst zwei Funktion komponiert und dann auf ein Argument anwendet, oder ob man die eine Funktion zuerst auf das Argument anwendet und die andere Funktion danach auf das Ergebnis.

Achtung: Die Klammern in der Definition b)iii sind wichtig, ansonsten kommt eine recht verrückte Funktion dabei heraus.

A5-3 ***Modul*** *`Data.Map`* Dr. Jost möchte zur Erbauung der Moral Bonuspunkte für Hausübungen verteilen. Zur Verwaltung dieser Bonuspunkte erstellt er folgendes Modul:

```
module Bonus (eintragPunkte, eintragAbschreiber, punkteAuslesen, leer) where

  import qualified Data.Map as Map

  data Bonus    = Punkte [Int] | Abschreiber
  type Student  = String -- zur Vereinfachung; sonst besser data verwenden
  type Register = Map.Map Student Bonus

  leer :: Register
  leer = undefined --TODO

  punkteAuslesen :: Student -> Register -> Int
  punkteAuslesen = undefined --TODO

  eintragAbschreiber :: Student -> Register -> Register
  eintragAbschreiber = undefined --TODO

  eintragPunkte :: Student -> Int -> Register -> Register
  eintragPunkte = undefined --TODO
```

Ein `Register` ordnet jedem `Student` einen `Bonus` zu. Ein `Bonus` ist entweder eine Liste von erzielten Hausaufgabenpunkten oder der Vermerk, dass der Student mindestens einmal abgeschrieben hat.

a) Vervollständigen Sie die Implementation durch Bearbeitung der vier mit –TODO— markierten Stellen. Die Funktion `punkteAuslesen` soll die Summe der erzielten Hausaufgabenpunkte für den abgefragten Studenten liefern; ist der Student unbekannt oder ein bekannter Abschreiber, so soll 0 geliefert werden. `eintragAbschreiber` soll den Eintrag des Studenten auf `Abschreiber` setzen; alle eventuell bis dahin erzielten Bonuspunkte verfallen. `eintragPunkte` fügt einem Eintrag eines Studenten die angegebenen Punkte hinzu; dabei sollen zur Protokollierung die Punkte nicht sofort aufaddiert werden, sondern die Liste mit Punkten einfach um einen Eintrag erweitert werden; unbekannte Studenten sollen neu angelegt werden; Abschreiber dürfen keine Punkte mehr sammeln und bleiben Abschreiber. *Beispiele:*

```
> let r1 = eintragPunkte      "Martin"  8 (eintragPunkte "Steffen" 4 leer)
> let r2 = eintragAbschreiber "Martin"    (eintragPunkte "Steffen" 6 r1)
> let r3 = eintragPunkte      "Martin" 12 (eintragPunkte "Steffen" 2 r2)
> punkteAuslesen "Steffen" r3
12
> punkteAuslesen "Martin" r3
0
```

LÖSUNGSVORSCHLAG:

```
module Bonus (eintragPunkte, eintragAbschreiber, punkteAuslesen, leer) where
  import qualified Data.Map as Map
  type Map      = Map.Map
  type Register = Map Student Bonus
  type Student  = String -- zur Vereinfachung; sonst besser data verwenden
  data Bonus    = Punkte [Int] | Abschreiber

  leer :: Register
  leer = Map.empty

  punkteAuslesen :: Student -> Register -> Int
  punkteAuslesen stud reg = case Map.lookup stud reg of
      (Just (Punkte punkte)) -> sum punkte
      (Just Abschreiber)     -> 0
      Nothing                -> 0

  eintragAbschreiber :: Student -> Register -> Register
  eintragAbschreiber stud reg = Map.insert stud Abschreiber reg

  eintragPunkte :: Student -> Int -> Register -> Register
  eintragPunkte stud pts reg = case Map.lookup stud reg of
    Nothing                -> Map.insert stud (Punkte [pts])        reg
    (Just Abschreiber)     ->                                       reg
    (Just (Punkte punkte)) -> Map.insert stud (Punkte (pts:punkte)) reg
```

Diese elemtare Lösung kann man noch vereinfachen, z.B. könnte man in `punkteAuslesen` die letzten beiden Fallunterscheidung zusammenfassen. Auch in der Vorlesung nicht vorgestellte Funktionen höherer Ordnung, wie z.B. `Data.Map.update` könnten hier nützlich sein. Eventuell könnte man auch mit Hilfsfunktion besser strukturieren, wie etwa

```
  addBonus :: Int -> Bonus -> Bonus
  addBonus p (Punkte pts) = Punkte (p:pts)
  addBonus _ Abschreiber  = Abschreiber

  getBonus :: Bonus -> Int
  getBonus (Punkte pts) = sum pts
  getBonus Abschreiber  = 0
```

Allerdings möchte man aus Effizienzgründen trotzdem noch den zweiten Fall in `eintragPunkte` behalten, welcher eine völlig unveränderte Map zurück gibt.

b) Warum werden in der ersten Zeile des oben vorgegebenen Codes vier Funktionsnamen in runden Klammern gelistet? Was wäre der Vorteil/Nachteil, wenn dort nur `module Bonus where` stünde?

LÖSUNGSVORSCHLAG:

Es handelt sich hier um eine Exportbeschränkung, d.h. außerhalb des Moduls sind nur diese vier Funktionen sichtbar. Damit dürfen z.B. die Konstruktoren `Punkte` und `Abschreiber` außerhalb nicht verwendet werden. Dies könnte man als Nachteil werten; es ist aber von Vorteil, wenn man später das Modul mal überarbeiten muss: da die Konstruktoren außerhalb des Moduls nicht verwendet werden können kann man diese beliebig abändern, ohne das Probleme für bestehenden Code außerhalb des Moduls entstehen.

H5-1 ***Falten*** (0 Punkte; Datei `H5-1.hs` als Lösung abgeben)
Schnell und ohne dabei in die Folien zu schauen: Definieren Sie die Funktionen `length` und `reverse` ohne direkte Rekursion, sondern nur unter Verwendung von `foldl`.
Zur Erinnerung: `foldl :: (b -> a -> b) -> b -> [a] -> b`

LÖSUNGSVORSCHLAG:

```
length1 :: [a] -> Int
length1 = foldl (\acc _ -> succ acc) 0

length2 :: [a] -> Int
length2 = foldl (const . succ) 0

length3 :: [a] -> Int
length3 = foldl (curry fst . succ) 0

reverse1 :: [a] -> [a]
reverse1 = foldl (flip (:)) []
```

`curry fst` ist praktisch identisch zu `const`

H5-2 ***Module*** (6 Punkte; Datei `Warteschlange.hs` abgegeben)
In der Vorlesung wurde das Modul `Data.Map` behandelt, welche eine abstrakte Datenstruktur für endliche Abbildungen bereitstellt.

Schreiben Sie ganz ähnlich dazu ein Modul `Warteschlange`, welches ausschließlich folgende Funktionen exportiert:

```
leer       :: Warteschlange a
einstellen :: a -> Warteschlange a -> Warteschlange a
abholen    :: Warteschlange a -> (Maybe a, Warteschlange a)
fanwenden  :: (a -> b) -> Warteschlange a -> Warteschlange b
```

a) Funktion `leer` liefert eine leere Warteschlange. Bei einer Warteschlange können wir mit `einstellen` Werte in die Warteschlange einfügen; und mit `abholen` das zuerst eingestellte Element wieder herausholen (First-In-First-Out).

Es ist Ihnen überlassen, wie Sie die Warteschlange innerhalb des Moduls tatsächlich implementieren, so lange alle vorgegebenen Typsignaturen eingehalten werden. Wichtig ist, dass der Datentyp abstrakt bleibt, d.h. es werden keine Konstruktoren, sondern nur die oben genannten Funktionen exportiert.

Für das Beispiel in der rechten Spalte haben wir lediglich zur Demonstration auch noch eine `Show`-Instanz definiert, welche von Ihnen aber nicht gefordert wird. Damit wird auch schon eine Möglichkeit verraten, eine Warteschlange effizient zu implementieren: Eine Warteschlange besteht aus zwei Listen; neue Elemente fügen wir immer in die erste Liste ein; das nächste auszulieferende Element ist der Kopf der zweiten Liste. Falls die zweite Liste leer ist, dann befüllen wir diese durch Umdrehen aller Elemente der ersten Liste. Sind beide Listen leer, dann liefert `abholen` einfach `(Nothing,leer)` zurück.

Sie dürfen hier alle Funktionen des Moduls `Data.Map` (und `Prelude`) verwenden; die auf den Folien 05-41ff. behandelten Funktion reichen aber bereits aus.

Beispiel:

```
> einstellen 'a' leer
WS("a","")
> einstellen 'b' it
WS("ba","")
> einstellen 'c' it
WS("cba","")
> abholen it
(Just 'a',WS("","bc"))
> abholen (snd it)
(Just 'b',WS("","c"))
> einstellen 'd' (snd it)
WS("d","c")
> abholen it
(Just 'c',WS("d",""))
> abholen (snd it)
(Just 'd',WS("",""))
> abholen (snd it)
(Nothing,WS("",""))
```

b) Implementieren Sie die Funktion `fanwenden`, welche eine Funktion auf alle Elemente in der Warteschlange anwendet. *Beispiele:*

```
> abholen (fanwenden (*2) (einstellen 5 (einstellen 3 (einstellen 1 leer))))
(Just 2,WS([],[6,10]))

> fanwenden fromEnum (einstellen 'B' (einstellen 'A' leer))
WS([66,65],[])
```

LÖSUNGSVORSCHLAG:

Wir simulieren die Warteschlange durch zwei Listen, um eine amortisierte konstante Laufzeit für beide Operationen einstellen/abholen zu erreichen. Jedes Element wird in der Warteschlange maximal 3 mal angefasst: beim Einstellen, beim Umdrehen und beim abholen.

Würde man nur eine einzelne Liste verwenden, dann müsste man entweder bei einstellen oder abholen die gesamte Warteschlange durchgehen; man hätte dann also eine lineare Laufzeit bei dieser einen Operation, welche alle Elemente in der Warteschlange jedes Mal alle anfassen müsste.

```
module Warteschlange (leer,einstellen,abholen,fanwenden) where

  data Warteschlange a = WS [a] [a]

  leer :: Warteschlange a
  leer = WS [] []

  einstellen :: a -> Warteschlange a -> Warteschlange a
  einstellen x (WS ein aus) = WS (x:ein) aus

  abholen :: Warteschlange a -> (Maybe a, Warteschlange a)
  abholen (WS ein (x:aus)) = (Just x, WS ein aus)
  abholen (WS []     [])   = (Nothing, leer)
  abholen (WS ein    [])   = abholen (WS [] (reverse ein))

  fanwenden :: (a -> b) -> Warteschlange a -> Warteschlange b
  fanwenden f (WS ein aus) = WS (map f ein) (map f aus)

  instance Functor Warteschlange where
    fmap f (WS ein aus) = WS (map f ein) (map f aus)

  instance Show a => Show (Warteschlange a) where
    show (WS ein aus) = "WS"++show (ein,aus)
```

H5-3 ***Abstiegsfunktion III*** (3 Punkte; Abgabeformat: Text oder PDF)

Beweisen Sie, dass folgende Funktion für alle nicht-leeren Listen terminiert, welche ausschließlich gerade Zahlen enthalten.

```
baz :: [Int] -> Int
baz (x1:x2:xs)
  | even x1, even x2  = baz (2*x1 : xs) + baz (xs ++ [2*x2])
  | otherwise         = baz (x1+x2 `div` 2 : 77 : xs)
baz [x1] = x1
baz []   = 42 `div` 0
```

Hinweis: Auf) und Def) sind in dieser Aufgabe besonders wichtig! Sie dürfen zur Vereinfachung annehmen, dass das l_1`++`l_2 nur Elemente enthält, welche in dem Argument l_1 und/oder l_2 vorkommen; und dass $|l_1$`++`$l_2| = |l_1| + |l_2|$ gilt.

LÖSUNGSVORSCHLAG:

Wir haben A' als Menge aller nicht-leeren Listen von geraden Zahlen, und damit insbesondere $A' \subsetneq$ `[Int]`. Wegen $A' \neq A$ müssen wir besonders auf Auf) und Def) achten!

Auf) Wir müssen also zeigen, dass alle Aufrufe nicht aus dieser Menge herausführen! Es gibt 3 rekursive Aufrufe:

- `baz (2*x1 : xs)` Nach Annahme enthält die Liste `xs` nur gerade Elemente, ist aber möglicherweise leer. `2*x1` ist immer eine gerade Zahl und da wir diese vor `xs` hängen, erfolgt der Aufruf also mit einer nicht-leeren Liste gerade Elemente; das Argument ist also in A'.
- `baz (xs ++ [2 * x2]` Nach Annahme enthält die Liste `xs` nur gerade Elemente, ist aber möglicherweise leer. `2*x2` ist immer eine gerade Zahl. Nach den erlaubten Annahmen über `(++)` ist das Ergebnis eine nicht-leere Liste mit geraden Elementen, also aus A': das zweite Argument ist eine nicht-leere Liste; beide Argumente enthalten nur gerade Zahlen.
- `baz (x1+x2 `div` 2 : 77 : xs)` Dieser Fall würde aus A' herausführen. Der Fall ist jedoch unproblematisch, da er nur Eintritt, wenn die Eingabeliste mindestens ein ungerades Element enthält. Dann war die Eingabe jedoch nicht aus A'.

Def) Die Fallunterscheidung durch Mustervergleiche deckt alle Fälle ab und ist unproblematisch. Die Fallunterscheidung durch Wächter im ersten Mustervergleich wird mit einem `otherwise` abgeschlossen und ist daher auch vollständig.

Problematisch können höchstens die Aufrufe von `div` sein, doch diese erfolgen nur für Aufrufe mit Argumenten, welche nicht aus A' sind: entweder enthielt die Eingabeliste eine ungerade Zahl, oder die Eingabeliste war leer.

Abst) Als Abstiegsfunktion verwenden wir die Länge der Eingabeliste, welche offensichtlich eine natürliche Zahl ist. Nach den vorherigen Betrachtungen müssen wir nur die beiden rekursiven Aufrufe im ersten Fall betrachten, da nur diese für Eingaben aus A' auftreten können. Sei die Eingabeliste eine Liste l mit $|l| = n$ und $n > 0$. Wegen dem Mustervergleich gilt sogar $n \geq 2$ und $|xs| = n - 2$.

Für den ersten Aufruf haben wir $|l| = n > n-1 = 1+(n-2) = 1+|$`xs`$| = |$`2*x1 : xs`$|$. Für den zweiten Aufruf rechnen wir $|l| = n > n-1 = (n-2)+1 = |$`xs`$| + |$`[2*x2]`$| = |$`xs ++[2*x2]`$|$.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den **2.06.2015**, 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.