Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

4. Musterlösung zur Vorlesung

Programmierung und Modellierung

A4-1 ***Instanzen von Typklassen definieren*** Gegeben sind folgende Deklarationen:

```
data List a = Leer | Element a (List a)  deriving Show

class Messbar a where
  messen :: a -> Double
```

a) Machen Sie den Datentyp `List a` zu einer Instanz der Klasse `Messbar`. Das Maß eines Wertes des Typs `List a` soll einfach die Länge der Liste sein.

LÖSUNGSVORSCHLAG:

Wir müssen also lediglich die Längen-Funktion implementieren:

```
instance Messbar (List a) where
  messen Leer = 0
  messen (Element _ l) = 1 + messen l
```

b) Machen Sie den Datentyp `List a` zu einer Instanz der Klasse `Eq`, falls auch `Eq a` gilt. Siehe dazu auch Folie 05-15. (Die Aufgabe einfach nur durch Hinzufügen von `deriving Eq` zu lösen ist hier natürlich nicht im Sinne der Aufgabenstellung.)

LÖSUNGSVORSCHLAG:

Die Lösung steht praktischerweise ja schon auf Folie 05-25, jetzt müssen wir diese nur noch verstehen:

```
instance Eq a => Eq (List a) where
  Leer           == Leer           = True
  (Element x xs) == (Element y ys) = (x == y) && (xs == ys)
  _              == _              = False
```

c) Vergleichen Sie die Funktion `compare :: Ord a => a -> a -> Ordering` von Folie 05-26 mit folgender Funktionsdefinition und finden Sie einen grundlegenden Unterschied:

```
vergleiche :: (Messbar a, Messbar b) => a -> b -> Ordering
vergleiche x y = messen x `compare` messen y
```

LÖSUNGSVORSCHLAG:

Die Funktion `compare` erlaubt nur den Vergleich von zwei Werten des gleichen Typs aus der Typklasse `Ord`, also z.B. `[a]` mit `[a]` zu vergleichen. Die Funktion `vergleiche` erlaubt dagegen den Vergleich von zwei Werten verschiedener Typen, so lange bei der Typklasse `Messbar` angehören, z.B. `List a` mit `Brotzeit` aus Aufgabe H4-3.

Weiterhin gibt es auch noch Unterschiede in der Bedeutung, so gilt z.B.:

```
> Element 9 Leer `vergleiche` Element 1 (Element 2 Leer)
LT
> [9] `compare` [1,2]
GT
```

A4-2 ***Induktion*** Gegeben ist folgende rekursive Funktionsdefinition:

```
foo :: (Integer,Integer) -> Integer
foo (n,m)  |  n == 0    = m+1
           |  n >  0    = foo (n-1, foo (n-1,m))
           |  otherwise = error "1. argument for foo must not be negative."
```

Beweisen Sie mit Induktion, dass für beliebige $(n, m) \in \mathbb{N} \times \mathbb{N}$ die Gleichung $\texttt{foo}(n, m) = 2^n + m$ gilt. Überlegen Sie sich zuerst worüber die Induktion geführt werden soll, also z.B. ob über n oder m oder $n + m$ oder $n \cdot m$ oder $\max(n, m)$ oder ...

LÖSUNGSVORSCHLAG:

Wir beweisen die Gleichung mit Induktion über n, da bei jedem rekursiven Aufruf n direkt kleiner wird. Die Aufgabenstellung verlangt den Beweis glücklicherweise nur für alle $(n, m) \in \mathbb{N} \times \mathbb{N}$, damit gilt schob $n \in \mathbb{N}$. Ansonsten müssten wir die Induktion über $\max(n, 0)$ oder ähnliches führen, da die Funktion ja Paaren von ganzen Zahlen als Eingabe verarbeitet und wir jedoch nur über geordnete Mengen mit einem kleinsten Element induzieren dürfen.

Induktionsanfang: Sei $n = 0$. Es gilt $\texttt{foo}(0, m) = m + 1 = 1 + m = 2^0 + m$ wie benötigt. Damit ist die Behauptung für den Basisfall bewiesen.

Induktionsschritt: Sei $n > 0$. Nehmen wir an, dass die Induktionshypothese für alle kleineren Werte als n gilt, d.h. wir dürfen die Gleichung $\texttt{foo}(x, y) = 2^x + y$ für alle $x < n$ verwenden.

Betrachten wir nun, was die Funktion `foo` an der Stelle (n, m) mit $n > 0$ macht. Nach der angegebenen Definition haben wir $\texttt{foo}(n, m) = \texttt{foo}(n - 1, \texttt{foo}(n - 1, m))$. Da $n - 1 < n$ gilt, dürfen wir die Induktionsannahme zwei Mal einsetzen:

$$\begin{aligned} \texttt{foo}(n, m) &= \texttt{foo}(n - 1, 2^{n-1} + m) \\ &= 2^{n-1} + (2^{n-1} + m) \end{aligned}$$

Jetzt müssen wir nur noch etwas umformen:

$$\begin{aligned} &= (2^{n-1} + 2^{n-1}) + m \\ &= (2 \cdot 2^{n-1}) + m \\ &= 2^n + m \end{aligned}$$

womit auch dieser Fall abgeschlossen wäre.

Damit haben wir mit Induktion also bewiesen: Für alle $n \in \mathbb{N}$ gilt $\texttt{foo}(n, m) = 2^n + m$. Da im Beweis keine Annahme über m gemacht wurde, gilt diese Formel sogar für alle $n \in \mathbb{N}$ und für alle $m \in \mathbb{N}$.

A4-3 ***Induktion mit Listen I*** Wir bezeichnen mit $|l| \in \mathbb{N}$ die Länge einer Liste l. Für eine beliebige nicht-leere Liste $(x : xs)$ gilt $|(x : xs)| = 1 + |xs|$.

a) Beweisen Sie mit Induktion über die Länge der Liste, dass für alle ganzen Zahlen $z \in \mathbb{Z}$, für alle natürlichen Zahlen $n \in \mathbb{N}$, und alle Listen von ganzen Zahlen zs mit $|zs| = n$ die Gleichung $|\texttt{insert z zs}| = 1 + |zs|$ gilt. Die Funktion `insert` ist definiert wie auf Folie 3-25:

```
insert :: Int -> [Int] -> [Int]
insert x   []  = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

Da Haskell eine rein funktionale Sprache ist, dürfen wir die definierenden Gleichung einer Funktion beliebig von links-nach-rechts oder auch von rechts-nach-links einsetzen. Lediglich bei überlappenden Mustervegleichen und/oder Wächtern müssen wir die entsprechenden Seitenbedingungen beachten. So dürfen wir zum Beispiel in unseren Rechnungen den Ausdruck $y : \texttt{insert}\ x\ ys$ nur dann durch $\texttt{insert x}\ (\texttt{y} : \texttt{ys})$ ersetzen, wenn $x > y$ angenommen werden darf. Das ist eigentlich klar, denn ansonsten würde ja bei der Ausführung der andere Zweig ausgewählt.

LÖSUNGSVORSCHLAG:

Es seien $n \in \mathbb{N}$ und $z \in \mathbb{Z}$ feste aber beliebige Zahlen und zs eine beliebige Liste von ganzen Zahlen mit $|zs| = n$. Wir beweisen $|\texttt{insert z zs}| = 1 + |zs|$ mit Induktion über n.

Für den Induktionsanfang haben wir $n = 0$. Die einzige Liste mit Länge 0 ist die leere Liste. Nach der definierenden Gleichung für `insert` gilt direkt:

$$|\texttt{insert}\ z\ []| = |[z]| = 1 = 1 + 0$$

Es sei nun $n > 0$. Damit ist zs nicht leer. Bezeichnen wir das erste Element von zs mit v und den Rest der Liste mit vs, also $zs = v : vs$. Es gilt $|vs| = n - 1$.

Jetzt müssen wir eine Fallunterscheidung durchführen. Falls $z \leq v$ gilt, so rechnen wir:

$$|\texttt{insert}\ z\ (v : vs)| = |z : v : vs| = 1 + 1 + |vs| = 2 + (n - 1) = 1 + n$$

Im anderen Fall gilt $z > v$ und wir setzen entsprechend ein:

$$|\texttt{insert}\ z\ (v : vs)| = |v : \texttt{insert}\ z\ vs| = 1 + |\texttt{insert}\ z\ vs| = 1 + (1 + |vs|) = 2 + (n - 1) = 1 + n$$

Die erste Gleichung folgt nach Definition. Die dritte Gleichung gilt wegen der Induktionshypothese, welche wir annehmen dürfen, da $|vs| < n$ gilt. Alle anderen Gleichung sind elementare Rechnenschritte.

In beiden Fällen haben wir das benötigte Ergebnis $1 + n$ erhalten, was den Beweis vollendet.

b) Beweisen Sie, dass für eine beliebige Liste l von ganzen Zahlen gilt

$$|l| = |\texttt{sort}\ l|$$

Dabei dürfen Sie die Aussage von Teilaufgabe a) ohne weiteres direkt verwenden. Die Funktion `sort` ist definiert wie auf Folie 3-25.

```
sort :: [Int] -> [Int]
sort    []  = []
sort (x:xs) = insert x (sort xs)
```

LÖSUNGSVORSCHLAG:

Es seien $n \in \mathbb{N}$ und $x \in \mathbb{Z}$ feste aber beliebige Zahlen und l eine beliebige Liste von ganzen Zahlen mit $|l| = n$. Wir beweisen $|l| = \texttt{sort l}$ mit Induktion über n.

Der Induktionsanfang $n = 0$ ist einfach, denn die einzige Liste mit Länge 0 ist die leere Liste. Nach Definition der Funktion gilt direkt $|\texttt{sort}\ [\,]| = |[\,]| = 0$.

Es sei nun $n > 0$. Damit ist l nicht leer. Bezeichnen wir das erste Element von l mit x und den Rest mit xs, also $l = x : xs$. Es gilt $|xs| = n - 1$. Wir rechnen durch Einsetzen der definierenden Funktionsgleichungen:

$$|\texttt{sort}\ (x : xs)| = |\texttt{insert}\ x\ (\texttt{sort}\ xs)| = 1 + |\texttt{sort}\ xs| = 1 + (n - 1) = n$$

Die erste Gleichung gilt nach der Definition von `sort`. Die zweite Gleichung folgt aus dem Lemma aus Teilaufgabe a). Da wir dieses Lemma bereits bewiesen haben, können wir dieses für beliebige Listen von beliebiger Länge anwenden. Die dritte Gleichung folgt nach Induktionsannahme, welche wir wegen $|xs| < n$ anwenden dürfen.

A4-4 *Fehlerhafte Induktion* Durch vollständige Induktion wollen wir zeigen:

Alle Smartphones benutzen das gleiche Betriebssystem.

Um Induktion einsetzen zu können, verallgemeinern die Aussage zu *"In einer Menge von n Smartphones benutzen alle das gleiche Betriebssystem."* Da die Anzahl aller Smartphones in der Welt eine natürliche Zahl ist, reicht dies für die ursprüngliche Aussage.

Der Induktionsanfang ist klar: Ein Smartphone benutzt das gleiche Betriebssystem wie es selbst.

Angenommen, wir hätten eine Menge von $n+1$ Smartphones. Wählen wir irgendein Smartphone davon aus und bezeichnen es mit s. Die übrigen Smartphones bilden eine Menge von n Smartphones. Nach Induktionsvoraussetzung benutzen diese n Smartphones alle das gleiche Betriebssystem. Nun entfernt man von diesen n Smartphones mit gleichem Betriebssystem eines und fügt s wieder dazu. Damit haben wir wieder eine Menge mit n Smartphones, welche nach Induktionsvoraussetzung wieder alle das gleiche Betriebssystem besitzen – insbesondere also auch s. Damit benutzen aber offenbar alle $n+1$ Smartphones das gleiche Betriebssystem und die Behauptung ist bewiesen.

Wo liegt der Fehler? Begründen Sie Ihre Antwort!

LÖSUNGSVORSCHLAG: Die Verallgemeinerung vor dem eigentlichen Beweis ist unproblematisch. In der Argumentation wird die Induktionsvoraussetzung korrekt verwendet. Jedoch wird beim zweiten Aussondern implizit davon ausgegangen, dass die Menge der verbleibenden $n-1$ Smartphones nicht leer ist. (Dies ist die Schnittmenge der beiden Mengen, für welche die Induktionsvoraussetzung verwendet wird.) Doch ein beliebiges Smartphone aus dieser Menge wird benötigt, um zu zeigen, dass $n+1$ Smartphones das gleiche Betriebssystem benutzen. Das Problem liegt also beim Fall $n+1=2$, also $n=1$.

H4-1 ***Wiederholung: Listen, Rekursion*** (0 Punkte; Datei `H4-1.hs` als Lösung abgeben)
Implementieren Sie schnell die Funktion `concat :: [[a]] -> [a]` aus der Standardbibliothek, welche die Elemente einer Liste von Listen miteinander verkettet. Verwenden Sie dazu Pattern-Matching auf Listen und Rekursion. Zur Vermeidung von Namenskonflikten soll die Funktion `myConcat` heissen! Sie dürfen die Infix-Funktion `(++) :: [a] -> [a] -> [a]` verwenden. *Beispiele:*

```
> myConcat [[1,2,3],[4,5],[],[6],[7,8]]   |   > myConcat ["Hi ", "there", "!"]
[1,2,3,4,5,6,7,8]                         |   "Hi there!"
```

LÖSUNGSVORSCHLAG:

```
myConcat :: [[a]] -> [a]
myConcat (  []) = []
myConcat (l:ls) = l ++ myConcat ls
```

Unsere Eingabe ist eine Liste, also unterscheiden wir zuerst per Pattern-Matching ob die Eingabeliste leer ist, oder nicht. Ist die Eingabeliste nicht leer, dann müssen wir das erste Element laut Aufgabenstellung verketten, also `(++)` anwenden; doch was ist das zweite Argument für diese Verkettung? Das zweite Element ist die Verkettung der restlichen Elemente, welche wir rekursiv durch die Anwendung von `myConcat` erhalten. Wir verketten also das erste Element der Eingabeliste mit der Verkettung der restlichen Elemente der Eingabeliste.

Die runden Klammern im zweiten Match sind notwendig, im ersten aber nicht. Denn `(:)` ist ein Infix-Konstruktor mit zwei Argumenten, während `[]` ein Konstruktor ohne Argumente ist.

Eine endrekursive Version ist hier nicht möglich (bzw. unsinnig, da `(++)` nicht endrekursiv formuliert werden kann, wenn die Reihenfolge der Elemente beibehalten werden soll.

H4-2 ***Induktion mit Listen II*** (4 Punkte; Abgabeformat: Text oder PDF)
Es sei vs und ws zwei beliebige Listen, weiterhin sei $n = |vs|$. Beweisen Sie mit Induktion über die Länge $n \in \mathbb{N}$ von vs, dass $|\texttt{zip}\ vs\ ws| = \min\big(|vs|, |ws|\big)$ gilt, wobei `zip` definiert ist wie auf Folie 03-26. *Hinweis:* Eine Induktion über n reicht aus. Eine Induktion über die Länge von ws ist nicht notwendig!

```
zip :: [a] -> [b] -> [(a,b)]
zip    []     _   = []
zip    _      []  = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

LÖSUNGSVORSCHLAG:

Für den Induktionsanfang haben wir wie immer $n = 0$. Nach den Voraussetzungen gilt dann $|vs| = 0$ Nach der Definition von `zip` tritt dann unabhängig von ws immer der erste Fall ein und wir erhalten als Ergebnis die leere Liste, welche die Länge 0 hat. Es gilt also $|\texttt{zip}\ []\ ws| = |[]| = 0 = \min(0, |ws|)$ wie benötigt, wobei die letzte Gleichung aus $|ws| \geq 0$ folgt.

Es sei nun also $n > 0$. Damit ist vs also nicht leer. Jetzt benötigen wir eine weitere Fallunterscheidung.

Falls ws die leere Liste ist, dann tritt der zweite Fall der definierenden Gleichungen von `zip` ein. Es gilt also $|\texttt{zip}\ vs\ []| = |[]| = 0 = \min(|vs|, 0)$ wie benötigt, wobei die letzte Gleichung wegen $|vs| > 0$ folgt.

Falls ws nun ebenfalls nicht leer ist, so wählen ohne Beschränkung der Allgemeinheit folgende Bennung: $vs = x : xs$ und $ws = y : ys$. Es tritt der dritte Fall der definierenden Gleichungen von `zip` ein. Wir rechnen

$$\begin{aligned} |\texttt{zip}\ vs\ ws| &= |\texttt{zip}\ (x : xs)\ (y : ys)| \\ &= |(x, y) : \texttt{zip}\ xs\ ys| \\ &= 1 + |\texttt{zip}\ xs\ ys| \end{aligned}$$

Wegen $|vs| > |xs|$ dürfen wir die Induktionshypothese einsetzen, also gilt $|\texttt{zip}\ xs\ ys| = \min\big(|xs|, |ys|\big)$.

$$\begin{aligned} &= 1 + \min\big(|xs|, |ys|\big) \\ &= \min\big(1 + |xs|, 1 + |ys|\big) \\ &= \min\big(|x : xs|, |y : ys|\big) \\ &= \min\big(|vs|, |ws|\big) \end{aligned}$$

H4-3 ***Instanzen II*** (4 Punkte; Datei `H4-3.hs` als Lösung abgeben)
Gegeben sind folgende Deklarationen:

```
data Brotzeit = Leberkas | Weisswurscht Int | Breze Brotzeit
  deriving (Show, Eq)

class Messbar a where
  messen :: a -> Double
```

a) Machen Sie den Typ `Brotzeit` zur Instanz der Klasse `Messbar`. Jedes Lebensmittel zählt einmal; also der Leberkas hat das Maß 1, n Weisswürste haben das Maß $\frac{n}{2}$, und eine Breze mit Brotzeit hat das Maß 1 plus der beinhalteten Brotzeit.

LÖSUNGSVORSCHLAG:

Hier ist nicht wirklich viel zu tun, als den Text direkt in Code umzusetzen:

```
instance Messbar Brotzeit where
  messen Leberkas = 1
  messen (Weisswurscht n) = fromIntegral n / 2
  messen (Breze b) = 1 + messen b
```

b) Machen Sie den Typ `Brotzeit` zu einer Instanz der Klasse `Ord`. Eine Brotzeit umso "größer", je mehr Brezen diese enthält. Eine Leberkas ist genausogut wie 2 Weisswürschte, aber 3 Weisswürschte schlagen jeden Leberkas. Siehe dazu auch Folie 05-26. *Beispiele:*

```
> compare (Breze Leberkas) (Breze (Weisswurscht 1))
GT
> compare (Breze (Weisswurscht 7)) (Breze (Breze Leberkas))
LT
> compare (Weisswurscht 3) Leberkas
GT
```

LÖSUNGSVORSCHLAG:

Hier müssen wir sorgfältig alle Fälle durchgehen:

```
instance Ord Brotzeit where
  compare Leberkas Leberkas = EQ
  compare (Weisswurscht n) (Weisswurscht m) = compare n m
  compare (Breze x) (Breze y) = compare x y
  compare (Breze x) _ = GT
  compare _ (Breze y) = LT
  compare Leberkas (Weisswurscht w) = compare 2 w
  compare (Weisswurscht w) Leberkas = compare w 2
```

Hinweis: Die 5. Übung findet nächste Woche regulär statt. Übungsblatt 6 wird aufgrund der Feiertage am Do 28.5., Fr 29.5., Di 2.6. und Mi 3.6. behandelt werden. Es entfallen lediglich die Übungen am Mi 27.5. und Fr 5.6. außerplanmäßig.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 19.05.2015, 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.