Dr. Steffen Jost
Leah Neukirchen

Ludwig-Maximilians-Universität München
Institut für Informatik
15. Mai 2018

Lösungsvorschlag zur 05. Übung zur Vorlesung

Programmierung und Modellierung

Hinweis: Aufgrund des Feiertages entfallen die Übungen am Dienstag, 22.5.18. Übungsblatt 6 wird am 23.5.18 erscheinen.

A5-1 ***Instanzen I*** Gegeben sind folgende Deklarationen:

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

class ToDouble a where
  toDouble :: a -> Double
```

a) Machen Sie den Datentyp `MyList a` zu einer Instanz der Klasse `ToDouble`. Der Doublewert einer Liste des Typs `MyList a` soll einfach die Länge der Liste sein.

LÖSUNGSVORSCHLAG:

Wir müssen also lediglich die Längen-Funktion implementieren:

```
instance ToDouble (MyList a) where
  toDouble Leer = 0
  toDouble (Element _ l) = 1 + toDouble l
```

b) Machen Sie den Datentyp `MyList a` zu einer Instanz der Klasse `Eq`, falls auch `Eq a` gilt. Siehe dazu auch Folie 5.28. (Die Aufgabe einfach nur durch Hinzufügen von `deriving Eq` zu lösen ist hier natürlich nicht im Sinne der Aufgabenstellung.)

LÖSUNGSVORSCHLAG:

Die Lösung steht ja eigentlich schon auf Folie 5.28, jetzt müssen wir diese nur noch verstehen und für den gegebenen Typen umsetzen:

```
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 5.29 mit folgender Funktionsdefinition und finden Sie einen grundlegenden Unterschied:

```
vergleiche :: (ToDouble a, ToDouble b) => a -> b -> Ordering
vergleiche x y = toDouble x `compare` toDouble 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 `ToDouble` angehören.

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
```

A5-2 ***Marktschreier*** Ein hessischer Marktschreier bietet auf den Viktualienmarkt in München zwei Sorten von Obst und Gemüse in fertig abgepackten Papiertüten an. Da die bayrischen Kunden nicht verstehen, was beworben wird, und es in den Papiertüten auch nicht sehen können, schüttet der verzweifelte Marktschreier die verschiedenen Produkte in immer größere Mischtüten um. Wir modellieren das wie folgt:

```
data Sordde = Erwel | Gummern                                  deriving (Eq, Show)
data Dudde = Dudd      { sort:: Sordde, kilo :: Double, duddepreis :: Double}
           | Gemischde { kilo :: Double, kilopreis  :: Double } deriving Show
```

Wir benötigen nun eine Funktion, welche uns das Zusammenschütten der Portionen ermöglicht. Dazu möchten wir die Klasse `Data.Monoid` aus der Standardbibliothek[1] verwenden:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Aufgabe: Machen Sie den Datentyp `Dudde` zu einer Instanz der Klasse `Monoid`! Die Konstante `mempty` modelliert einfach eine leere Tüte. Die Funktion `mappend` modelliert das Zusammenschütten. Bei zwei Tüten mit gleicher Sorte muss `mappend` also nur Preis und Gewicht addieren. Bei Mischtüten ist jedoch anstatt dem absoluten Preis der Kilopreis anzugeben.

Beispiele:

```
> mappend (Dudd Erwel 4 7) (Dudd Erwel 2 3)
Dudd {sort = Erwel, kilo = 6.0, duddepreis = 10.0}
> (Dudd Gummern 2 1) `mappend` (Dudd Gummern 3 2)
Dudd {sort = Gummern, kilo = 5.0, duddepreis = 3.0}
```

[1] Bei älteren ghc Versionen muss man das Modul `Data.Monoid` explizit importieren (siehe Folie 5.41)

```
> mappend (Dudd Erwel 1 2) (Dudd Gummern 3 1)
Gemischde {kilo = 4.0, kilopreis = 0.75}
> mappend (Dudd Erwel 1 2) (Gemischde 4 0.75)
Gemischde {kilo = 5.0, kilopreis = 1.0}
> mappend (Gemischde 4 4) (Gemischde 4 2)
Gemischde {kilo = 8.0, kilopreis = 3.0}
```

Hinweis: Achten Sie darauf, die Invarianten der Klasse nicht zu verletzen. Hier gibt es drei Regeln:

- mappend mempty x = x
- mappend x mempty = x
- mappend x (mappend y z) = mappend (mappend x y) z

Es gilt also z.B.

```
> mappend (Dudd Erwel 1 2) mempty
Dudd {sort = Erwel, kilo = 1.0, duddepreis = 2.0}
> mappend mempty (Dudd Gummern 3 1)
Dudd {sort = Gummern, kilo = 3.0, duddepreis = 1.0}
```

LÖSUNGSVORSCHLAG:

```
instance Monoid Dudde where
  mempty = Gemischde { kilo=0, kilopreis=1 }

  mappend d1 (Gemischde 0 1) = d1 -- Sonderfall für mempty
  mappend d1 d2
    | kilo d1 <= 0 = d2
    | kilo d2 <= 0 = d1
  mappend (Gemischde k1 kp1) (Gemischde k2 kp2)
    = Gemischde (k1+k2) ((k1*kp1 + k2*kp2) / (k1+k2))
  mappend (Dudd s1 k1 p1) (Dudd s2 k2 p2)
    | s1==s2 = Dudd s1 (k1+k2) (p1+p2)
  mappend d1 d2 = mappend (mische d1) (mische d2)
    where
      mische :: Dudde -> Dudde
      mische (Dudd s1 k1 p1) = Gemischde {kilo=k1, kilopreis = (p1/k1)}
      mische annereDut = annereDut
```

Zu Demonstrationszwecken verwenden wir hier nur gelegentlich die Record-Syntax; sinnvoller wäre eine einheitliche Syntax innerhalb einer Definition.

Versionshinweis: Mit ghc Version 8.4.x wurde die Definition der Klasse `Data.Monoid` wie folgt geändert:

```
class Semigroup a => Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mappend = (<>)
class Semigroup a where
  (<>) :: a -> a -> a
```

D.h. mathematisch korrekt ist ein Monoid nun eine Halbgruppe mit neutralem Element. *In einfacheren Worten:* die Funktion `mappend` in eine eigene Typklasse ausgelagert und dort zur einer Infix-Funktion `(<>)` umbenannt.

Folgender Code funktioniert mit *beiden* Versionen, wobei man sich ab Version 8.4.x die zwei markierten Zeilen sparen kann:

```
import Data.Semigroup            -- nur für Versionen <= 8.2.x

instance Monoid Dudde where
  mempty  = Gemischde { kilo=0, kilopreis=1 }
  mappend = (<>)                 -- nur für Versionen <= 8.2.x

instance Semigroup where
  (<>) d1 (Gemischde 0 1) = d1   -- Sonderfall für mempty
  (<>) d1 d2
    | kilo d1 <= 0 = d2
    | kilo d2 <= 0 = d1
  (<>) (Gemischde k1 kp1) (Gemischde k2 kp2)
    = Gemischde (k1+k2) ((k1*kp1 + k2*kp2) / (k1+k2))
  (<>) (Dudd s1 k1 p1) (Dudd s2 k2 p2)
    | s1==s2 = Dudd s1 (k1+k2) (p1+p2)
  (<>) d1 d2 = (<>) (mische d1) (mische d2)
    where
      mische :: Dudde -> Dudde
      mische (Dudd s1 k1 p1) = Gemischde {kilo=k1, kilopreis = (p1/k1)}
      mische annereDut = annereDut
```

A5-3 ***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 Smart-

phone 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 $n=1$.

Ende der Lösungsvorschläge für die Präsenzaufgaben.
Lösungsvorschläge für die Hausaufgaben folgen nach Ende der Abgabezeit.

Hinweise:

- Ab sofort dürfen alle Funktionen des Moduls `Prelude` verwendet werden, sofern in einer Aufgabe nichts anderes angegeben wurde.
- Nicht kompilierende Programmieraufgaben werden nicht korrigiert.
- Wir empfehlen in ghci ab sofort `:unset -XMonomorphismRestriction`, ggf. `ghci.conf` anpassen.

H5-1 ***Instanzen II*** (2 Punkte; Datei `H5-1.hs` als Lösung abgeben)
Gegeben sind folgende Deklarationen:

```
class ToDouble a where
  toDouble :: a -> Double

data Op = Plus | Minus | Mal | Geteilt
  deriving (Show, Eq)
data RechenBaum = BlattWert Double | AstOp RechenBaum Op RechenBaum
  deriving Show
```

Machen Sie den Datentyp `RechenBaum` zu einer Instanz der Klasse `ToDouble`. Der Doublewert eines Baumes des Typs `RechenBaum` entspricht für Blätter dem gespeicherten Double-Wert, und für Äste sind die Werte der Teilbäume mit der gegebenen Rechenoperation zu verknüpfen. *Beispiele:*

```
> toDouble (AstOp (AstOp (BlattWert 3) Plus (BlattWert 4)) Geteilt (BlattWert 5))
1.4
> toDouble (AstOp (BlattWert 2) Minus (AstOp (BlattWert 3) Mal (BlattWert 4)))
-10.0
```

H5-2 ***Modul*** *`Data.Map`* (2 Punkte; Datei `Bonus.hs` abgeben)
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
```

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?

H5-3 ***Module*** (2 Punkte; Datei `Warteschlange.hs` abgegeben)
In der Vorlesung wurde als Beispiel einer *abstrakten Datenstruktur* das Modul `Data.Map` behandelt. Implementieren Sie selbst eine abstrakte Datenstruktur in einem Modul `Warteschlange`, welches ausschließlich folgende Funktionen exportiert:

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

Funktion `leer` liefert eine leere Warteschlange. Bei einer Warteschlange können wir mit `einstellen` Werte in die Warteschlange hinten anstellen; 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 auszuliefernde 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.

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("",""))
```

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 26.5.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).