6. Aufgabenblatt zu Funktionale Programmierung vom Mi, 21.11.2018.
Fällig: Mi, 28.11.2018 (15:00 Uhr)

Themen: *Rechnen mit Funktionen, Funktionen als Argumente und Resultate von Funktionen, Funktionstransformatoren*

Zur Frist der Zweitabgabe: Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Homepage der LVA.

Aufgabe

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe6.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also wieder ein ‘gewöhnliches’ Haskell-Skript schreiben.

Versehen Sie wieder wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung benötigen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und (Wertvereinbarungen für) Konstanten.

- Schreiben Sie eine Haskell-Rechenvorschrift höherer Ordnung `map_2` mit Signatur:

  ```
  map_2 :: (a -> b) -> (a -> b) -> [a] -> [b]
  ```

  Angewendet auf zwei Funktionen f und g und eine Liste von Werten xs, werden f und g beginnend mit f abwechselnd auf die Elemente von xs angewendet, d.h., f wird auf das 1., 3., 5., usw. Listenelement angewendet, g auf das 2., 4., 6., usw. Element. Die leere Werteliste wird ident abgebildet.

  Überlegen Sie sich geeignete Testfälle (ohne Abgabe!), anhand derer Sie die Arbeitsweise Ihrer Implementierung überprüfen können. Denken Sie daran, für die Typvariablen `a` und `b` nicht nur Werte nichtfunktionaler Typen wie `Int`, `String`, `Bool`, `(String,Int)`, `BinTree`, `PolyBinTree c`, etc. zu wählen, sondern auch Werte funktionaler Typen wie `String -> Int -> Char`, `(PolyBin Tree c) -> Bool`, etc. Finden Sie z.B. passende Funktions- und Listenwerte `f,g :: (Int -> (Int -> Int))` und `ns :: [Int]` als Testdaten für die sich daraus bei Aufruf ergebende `map_2`-Instanz:

  ```
  map_2 :: (Int -> (Int -> Int)) -> (Int -> (Int -> Int)) -> [Int] -> [Int -> Int]
  ```

  Wenden Sie `map_n` auf `(map_2 f g ns)` und `ns` an: `map_n (map_2 f g ns) ns`. Welchen Typ hat der Wert dieses Ausdrucks? Welchen Wert hat er für die Wahl von `f` und `g` als Binomial- und Größter-gemeinsamer-Teiler-Funktion und von `ns` als Liste `[45,120,49]`? Welchen Typ und Wert haben die Ausdrücke `map_n (map_2 f g ns) [fac 3,3*fib 7-3,g (fac 4) (fib 5 + fib 7)]` und `map_n ((9.42/) : (map_2 (**) logBase) [100.0,100.0]) [3.0,3.0,1000000.0]`?

- Schreiben Sie eine Haskell-Rechenvorschrift höherer Ordnung `map_n` mit Signatur:

  ```
  map_n :: [(a -> b)] -> [a] -> [b]
  ```

  Angewendet auf eine Liste von Funktionen und eine Liste von Werten wendet `map_n` die Funktionen des ersten Arguments zyklisch (engl. *round robin fashion*) auf die Elemente der Liste von Werten an, d.h., die erste Funktion wird auf das erste Element der Wertliste angewendet, die zweite Funktion auf das zweite Element usw. Ist das zweite Argument kürzer als das erste Argument, kommen einige Funktionen nicht zum Zuge; ist das erste Element kürzer als das zweite, wird die zyklische Anwendung der Funktionen des ersten Arguments schlagend.

  Überlegen Sie sich geeignete Testfälle (ohne Abgabe!), anhand derer Sie die Arbeitsweise Ihrer Implementierung überprüfen können. Denken Sie auch hier an Werte funktionaler Typen.

- Der *Algorithmus von Euklid* zur Berechnung des *größten gemeinsamen Teilers* zweier natürlicher Zahlen m und n, $m, n \in IN_1$, ist wie folgt:

  > Wähle x gleich m und y gleich n. Ziehe wiederholt den kleineren der Werte von x und y vom größeren ab. Höre auf, wenn x und y denselben Wert haben. Dieser Wert ist der größte gemeinsame Teiler von m und n.

  Die folgende Haskell-Rechenvorschrift setzt diese Beschreibung um:

```
type Nat1 = Integer

ggt_euklid :: Nat1 -> (Nat1 -> Nat1)
ggt_euklid x y
 | x == y = x
 | x > y  = ggt_euklid (x-y) y
 | x < y  = ggt_euklid x (y-x)
```

  Zwei *Aufrufbeispiele* zur Illustration der Arbeitsweise von `ggt_euklid`:

```
m = 18
n = 12
ggt_euklid m n ->> ggt_euklid 18 12 ->> ggt_euklid 6 12 ->> ggt_euklid 6 6 ->> 6

mm = 20
nn = 35
ggt_euklid mm nn ->> ggt_euklid 20 35 ->> ggt_euklid 20 15 ->> ggt_euklid 5 15
                 ->> ggt_euklid 5 10 ->> ggt_euklid 5 5 ->> 5
```

  Obwohl `ggt_euklid` streng genommen eine Funktion höherer Ordnung ist, da sie natürliche Zahlen auf Funktionen natürlicher Zahlen in sich abbildet (was durch die explizit angegebene Klammerung in der Funktionssignatur betont wird), macht die saloppe Sprechweise von `ggt_euklid` als zweistelliger Funktion, die Paare natürlicher Zahlen auf natürliche Zahlen abbildet, deutlich, dass die Eigenschaft von `ggt_euklid` als Funktion höherer Ordnung auf einem formalen Argument beruht.

  In dieser Aufgabe wollen wir eine echt funktionale Implementierung des Algorithmus von Euklid erreichen, in dem Funktionen mittels eines Funktionentransformators auf Funktionen abgebildet werden. Argument- und Resultatfunktionen des Funktionentransformators sind dabei Zustandsfunktionen, der Funktionentransformator ist deshalb genauer ein Zustandsfunktionentransformator oder kürzer, ein Zustandstransformator.

  Die schrittweise Auswertung von `ggt_euklid` für `m` und `n` und `mm` und `nn` macht die Folge von Werten deutlich, die die Parameter `x` und `y` bei den verschiedenen Aufrufen von `ggt_euklid` haben:

  $\langle \mathtt{x} \leftarrow 18, \mathtt{y} \leftarrow 12 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 12 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 6 \rangle$

  $\langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 35 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 15 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 15 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 10 \rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 5 \rangle$

  Zustände sind Abbildungen von Variablen auf Werte. Wir können deshalb die Paare der Form $\langle \mathtt{x} \leftarrow 18, \mathtt{y} \leftarrow 12 \rangle$ als Repräsentationen von Zuständen über der zweielementigen Menge von Variablen $\{\mathtt{x},\mathtt{y}\}$ in die Menge natürlicher Zahlen IN_1 auffassen und die Übergangsrelation $\rightsquigarrow$ als Zustandstransformator, der Zustände auf Zustände abbildet:

$$\begin{aligned} \text{Zustandsmenge } \Sigma\text{:} \quad & \Sigma =_{df} \{\sigma \mid \sigma : \{\mathtt{x},\mathtt{y}\} \to IN_1\} \\ \text{Zustandstransformator } \rightsquigarrow\text{:} \quad & \rightsquigarrow : \Sigma \to \Sigma \end{aligned}$$

  In Haskell können wir Σ und $\rightsquigarrow$ folgendermaßen modellieren:

```
type Nat1                = Integer
data Variable            = X | Y deriving (Eq,Ord,Enum,Show)
type Variablen           = Variable
type Zustand             = (Variablen -> Nat1)
type Zustandsmenge       = Zustand
```

```
type Sigma                  = Zustandsmenge
type Zustandstransformator = (Sigma -> Sigma)

wellenpfeil :: Zustandstransformator
wellenpfeil ...
```

– Vervollständigen Sie die Implementierung von `wellenpfeil` im Sinn von Euklids Verfahrensidee; folgende Zustandsübergänge illustrieren dabei beispielhaft die Bedeutung des Zustandstransformators `wellenpfeil`:

$\langle \mathtt{x} \leftarrow 18, \mathtt{y} \leftarrow 12\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 12\rangle$
$\langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 12\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 6\rangle$
$\langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 6\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 6\rangle$

$\langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 35\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 15\rangle$
$\langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 15\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 15\rangle$
$\langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 15\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 10\rangle$
$\langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 10\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 5\rangle$
$\langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 5\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 5\rangle$

– Schreiben Sie mithilfe von `wellenpfeil` (oder der Abbildungsidee, auf der `wellenpfeil` beruht,) eine Funktion `ggt` zur Berechnung des größten gemeisamen Teilers zweier natürlicher Zahlen größer oder gleich 1 mit folgender Signatur:

```
ggt :: Zustandstransformator
```

Beachte: Anders als `ggt_euklid` ist der Zustandstransformator `ggt` eine echte (und nicht nur formal eine) Funktion höherer Ordnung, was die Auflösung der Signatur von `ggt` deutlich macht: `ggt :: (Variablen -> Nat1) -> (Variablen -> Nat1)`, d.h. `ggt` bildet Abbildungen auf Abbildungen ab.

– Wir bezeichnen die endlichen Zustandsübergangsfolgen, die in einem Zustand enden, in dem `x` und `y` denselben Wert haben, als *Berechnungsspuren* oder kurz als *Spuren*:

$\langle \mathtt{x} \leftarrow 18, \mathtt{y} \leftarrow 12\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 12\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 6, \mathtt{y} \leftarrow 6\rangle$

$\langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 35\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 20, \mathtt{y} \leftarrow 15\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 15\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 10\rangle \rightsquigarrow \langle \mathtt{x} \leftarrow 5, \mathtt{y} \leftarrow 5\rangle$

Schreiben Sie eine Haskell-Rechenvorschrift `ggt_spur` mit der Signatur:

```
type Spur = [Sigma]
ggt_spur :: Sigma -> Spur
```

Angewendet auf einen Zustand liefert `ggt_spur` die von diesem Zustand induzierte endliche Berechnungsspur. Zum Beispiel:

```
anfangszustand1 :: Sigma
anfangszustand1 = \z -> if z==X then 18 else 12
az1 = anfangszustand1
endzustand1 :: Sigma
endzustand1 = \z -> 6
ez1 = endzustand1

anfangszustand2 :: Sigma
anfangszustand2 = \z -> if z==X then 20 else 35
az2 = anfangszustand2
endzustand2 :: Sigma
endzustand2 = \z -> 5
ez2 = endzustand2

ggt_spur az1 ->>
 `[az1,wellenpfeil(az1),wellenpfeil(wellenpfeil(az1)),...,ez1]'  (symbolisch)
ggt_spur az2 ->>
 `[az2,wellenpfeil(az2),wellenpfeil(wellenpfeil(az1)),...,ez2]'  (symbolisch)
```

Überlegen Sie sich geeignete Testfälle (ohne Abgabe!), anhand derer Sie überprüfen können, dass `ggt_spur` tatsächlich Funktionslisten als Resultat liefert, d.h. Listen von Funktionen; konkret: Listen Zustände repräsentierender Funktionen.

- Schreiben Sie eine Haskell-Rechenvorschrift `zeige_spur` mit der Signatur:

  ```
  zeige_spur :: Spur -> String
  ```

  Angewendet auf eine Spur liefert `zeige_spur` eine Darstellung dieser Spur in Form einer Zeichenreihe, in der Funktionen durch ihre Argument/Bild-Paare dargestellt sind wie in folgenden Beispielen illustriert. Dabei stehen `-w->` und `<-` für ASCII-Approximationen von $\rightsquigarrow$ und $\leftarrow$. Vor und nach jedem Vorkommen von `-w->` steht genau ein Leerzeichen, weitere Leerzeichen werden nicht verwendet:

  ```
  zeige_spur (ggt_spur az1) ->>
   "[(x<-18,y<-12) -w-> (x<-6,y<-12) -w-> (x<-6,y<-6)]"
  zeige_spur (ggt_spur az2) ->>
   "[(x<-20,y<-35) -w-> (x<-20,y<-15) -w-> (x<-5,y<-15) -w-> (x<-5,y<-10) -w-> (x<-5,y<-5)]"
  ```

- Machen Sie den Typ `Spur'` zu einer Instanz der Typklasse `Show`: `newtype Spur' = Sp Spur`. Der Aufruf `show (Sp spur)` soll dabei wie bei `zeige_spur` erfolgen, lediglich die eckigen Klammern am Beginn und Ende der Aussgabe sollen entfallen wie nachstehend illustriert:

  ```
  show (Sp (ggt_spur az1)) ->>
   "(x<-18,y<-12) -w-> (x<-6,y<-12) -w-> (x<-6,y<-6)"
  show (Sp (ggt_spur az2)) ->>
   "(x<-20,y<-35) -w-> (x<-20,y<-15) -w-> (x<-5,y<-15) -w-> (x<-5,y<-10) -w-> (x<-5,y<-5)"
  ```

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 23.11.2018, werden wir uns u.a. mit der Aufgabe *Tortenwurf* beschäftigen.

Tortenwurf

Wir betrachten eine Reihe von $n + 2$ nebeneinanderstehenden Leuten, die von paarweise verschiedener Größe sind. Eine größere Person kann stets über eine kleinere Person hinwegblicken. Demnach kann eine Person in der Reihe so weit nach links bzw. nach rechts in der Reihe sehen bis dort jemand größeres steht und den weitergehenden Blick verdeckt.

In dieser Reihe ist etwas Ungeheuerliches geschehen. Die ganz links stehende 1-te Person hat die ganz rechts stehende $n + 2$-te Person mit einer Torte beworfen. Genau p der n Leute in der Mitte der Reihe hatten während des Wurfs freien Blick auf den Tortenwerfer ganz links; genau r der n Leute in der Mitte der Reihe hatten freien Blick auf das Opfer des Tortenwerfers ganz rechts.

Wieviele Permutationen der n in der Mitte der Reihe stehenden Leute gibt es, so dass gerade p von ihnen freie Sicht auf den Werfer und r von ihnen auf das Tortenwurfopfer hatten?

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl $n \leq 10$ von Leuten in der Mitte der Reihe, davon p mit $1 \leq p \leq n$ mit freier Sicht auf den Werfer und r mit $1 \leq r \leq n$ mit freier Sicht auf das Opfer, diese Anzahl von Permutationen berechnet.

Haskell Private

Die Anmeldung zu *Haskell Private* ist offen. Nutzen Sie die Möglichkeit! Auch die Möglichkeit, eigene Terminvorschläge zu machen. Nähere Hinweise und die URL zur Anmeldungsseite finden Sie auf der Homepage der Lehrveranstaltung.