LVA 185.A03 Funktionale Programmierung (WS 20)

Leit- und Kontrollfragen VI

Mi, 02.12.2020

Stoff: Vorlesungsteil VI – Kapitel 15, 16 und 17

Weiterführende Konzepte – Ein-/Ausgabe, Fehlerbehandlung, Module

(Ohne Abgabe, ohne Beurteilung; zur Selbsteinschätzung)

Teil VI, Kapitel 15 'Interaktive Programme: Ein- und Ausgabe'

1. *Referentielle Transparenz* ist eine grundlegende Eigenschaft rein funktionaler Programme.
   (a) Was bedeutet referentielle Transparenz?
   (b) Welche Probleme stellen sich für referentielle Transparenz im Zusammenhang mit Ein- und Ausgabe?
2. Wie werden in Haskell diese Probleme gelöst/vermieden?
3. Warum spielt der ansonsten weitgehend uninteressante Nulltupeltyp im Zuammenhang mit Ein- und Ausgabe eine bedeutsame Rolle?
4. Was unterscheidet Werte des vordefinierten polymorphen Datentyps `IO` von Werten aller anderen Typen in Haskell?
5. Die `do`-Notation in Haskell ist *syntaktischer Zucker*. Wofür?
6. Erklären Sie knapp, aber gut nachvollziehbar wie die Auswertung folgenden Ein-/Ausgabeprogramms vor sich geht:

```
summiere :: IO Int
summiere
 = do n <- getInt
      if n == 0
       then return 0
       else (do m <- summiere
                return (n + m))

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

   Annotieren Sie den Typ aller (Teil-) Ausdrücke.
7. Schreiben Sie die Programme `summiere` und `getInt` aus der vorigen Aufgabe ohne `do`-Notation.
8. Warum mag man folgendes Programm iterativartig nennen, auch wenn es nicht wirklich iterativ ist, keine while-Schleife darstellt, wie der Name suggerieren mag?

```
while :: IO Bool -> IO () -> IO ()
while bedingung aktion
  = do b <- bedingung
       if b
        then
         do aktion
            while bedingung aktion
        else
         return ()
```

Beschreiben Sie auch hier wieder knapp, aber gut nachvollziehbar, wie die Auswertung des Programms vor sich geht. Geben Sie auch hier wieder den Typ aller (Teil-) Ausdrücke an.

9. Woran erkennt man Ein-/Ausgabeausdrücke in Haskell?
10. Warum bezeichnet man Ein-/Ausgabeausdrücke in Haskell auch als Aktionen?
11. Wie lassen sich Ein-/Ausgabeaktionen zu Sequenzen zusammensetzen, komponieren?
12. Erklären Sie informell die operationelle Bedeutung von:
    (a) `(>>=) :: IO a -> (a -> IO b) -> IO b`
    (b) `(>>) :: IO a -> IO b -> IO b`
    (c) `return :: a -> IO a`
13. Veranschaulichen Sie die Bedeutung der Abbildungen aus der vorigen Aufgabe auch mithilfe graphischer Darstellungen.
14. Erläutern Sie den Unterschied von `return` in Haskell zu gleichbenannten Routinen aus anderen Sprachen, die Sie kennen.
15. Wie hängen die Abbildungen `(>>)` und `(>>=)` zusammen?
16. Wie hängen *unbenannte Bindungen* in `do`-Ausdrcuken und die Abbildung `(>>)` zusammen?
17. In welchem Sinn kann man `return` und `<-` (etwas salopp) als zueinander revers verstehen?
18. Lassen sich lokale Wertvereinbarungen in `do`-Ausdrücken einführen? Wenn ja, wie? Geben Sie ggf. ein illustrierendes Beispiel an.
19. Was ist das funktionale, was das prozedurale Verhalten eines Ein-/Ausgabeausdrucks?
20. Wie unterscheiden sich die Bedeutungen von
    (a) `<-` (in `do`-Ausdrücken)
    (b) `let` (in `do`-Ausdrücken)
    (c) `:=` (Wertzuweisungsoperator in imperativen Sprachen)

    voneinander?
21. Schreiben Sie jeweils
    (a) mit
    (b) ohne

    `do`-Notation ein Ein-/Ausgabeprogramm, dass
    - zur Aufgabe eines Zeichens auffordert
    - ein Zeichen einliest

    und diesen Zyklus so lange wiederholt, bis schließlich das Zeichen '`s`' für *stopp* eingegeben wird. Mit Ausnahme dieses Stoppzeichens '`s`' sollen anschließend alle gelesenen Zeichen zusammen mit einem kurzen die Ausgabe erklärenden Hinweis ausgegeben werden.
22. Ändern Sie Ihre Programme aus der vorigen Aufgabe so ab, dass nach Beendigung der Ausgabe statt aller gelesenen Zeichen nur die gelesenen Kleinvokale ausgegeben werden.
23. Ändern Sie Ihre Programme aus den beiden vorigen Aufgaben so ab, dass die Ein- und Ausgabe nicht mehr vom und auf den Bildschirm erfolgen, sondern von und in eine (andere) Datei.
24. Ändern Sie Ihre Programme aus den vorigen Aufgaben so ab, dass aus dem Eingabestrom von Zeichen die Ziffern herausgefiltert werden und nach Beendigung der Eingabe der Zahlwert dieser Ziffernfolge zusammen mit einem erklärenden Hinweis ausgegeben wird.
25. Ergänzen Sie für ein oder zwei der Programme aus der Voraufgabe zur Überprüfung der Typkorrektheit den Typ aller vorkommenden (Teil-) Ausdrücke.

Teil VI, Kapitel 16 'Robuste Programme: Fehlerbehandlung'

1. Welche
   (a) Ziele sollte eine systematische Fehlerbehandlung verfolgen?
   (b) Eigenschaften sollte sie dabei erfüllen?
2. Sind diese Ziele und Eigenschaften stets gleichzeitig zu erreichen oder stehen sie sich zum Teil im Wege? Begründen Sie Ihre Antwort.
3. Warum ist ein Weiterrechnen mit einem falschen/unsinnigen Wert oft gefährlicher als ein Fehlerabbruch?
4. Was ist mit dem Panikmodus als Fehlerbehandlung gemeint?
5. Wie lässt sich eine Fehlerbehandlung im Panikmodus in Haskell am einfachsten umsetzen?
6. Illustrieren Sie diese Umsetzung anhand eines passenden Beispiels.
7. Wo liegen die Stärken, Schwächen einer Fehlerbehandlung im Panikmodus?
8. Ein anderer Vorschlag zur Fehlerbehandlung empfiehlt die Verwendung von Auffangwerten.
   (a) Was ist damit gemeint?
   (b) Welche Varianten können möglicherweise genauer unterschieden werden?
9. Wie lässt sich eine Fehlerbehandlung mithilfe von Auffangwerten in Haskell umsetzen?
10. Illustrieren Sie diese Umsetzung anhand passendee Beispiele, ggf. je eines für verschiedene Varianten.
11. Wo liegen die Stärken, Schwächen einer Fehlerbehandlung mithilfe von Auffangwerten?
12. Illustrieren Sie die Stärken und Schächen einer Fehlerbehandlung mithilfe von Auffangwerten anhand geeigneter Beispiele.
13. Welche weitere Möglichkeit bietet sich in Haskell an, eine systematische Fehlerbehandlung durchzuführen?
14. Welches sind die Stärken und Schwächen dieser Möglichkeit?
15. Illustrieren Sie diese Möglichkeit wieder anhand treffender Beispiele.
16. Können Sie anhand Ihrer Beispiele auch die Stärken und Schwächen aufzeigen?
17. Was ist mit *Verschleierung* eines Fehlers durch Fehlerbehandlung gemeint?
18. Warum ist eine Verschleierung von Fehlern gefährlich?
19. Lässt sich eine Verschleierung von Fehlern stets vermeiden? Begründen Sie Ihre Antwort.
20. Welcher Typ ist in Haskell als `Fehlertyp` gemeint?
21. Wieviele Fehlerarten lässt dieser Typ unterscheiden?
22. Definieren Sie einen eigenen Fehlertyp in Haskell, der für `Int`-Werte zwischen den Fehlerarten *Überlauf*, *Unterlauf*, *Sonstiges* zu unterscheiden erlaubt.
23. In welcher Lesart kann man die Bedeutung der Funktion

    ```
    may_Maybe :: (a -> b) -> Maybe a -> Maybe b
    ```

    als ein 'Typ-Lifting' verstehen?
24. Welche andere Lesart gibt es für `map_Maybe` noch?
25. Wozu ist `map_Maybe` nützlich?

Teil VI, Kapitel 17 'Programmierung im Großen: Module'

1. Was bezeichnen die Begriffe

   (a) Kohäsion

   (b) Koppelung

   im Zusammenhang mit Modulen? Was beschreiben, wozu dienen sie?

2. Was haben Kohäsion und Koppelung mit *guter* Modularisierung zu tun?

3. Welche Perspektiven können wir einnehmen, um die *Güte* einer Modularisierung zu beurteilen?

4. Welche Eigenschaften sind unter diesen Perspektiven wichtig, um die *Güte* einer Modularisierung zu beurteilen?

5. Was sind Indizien einer wenig(er) gut gelungenen Modularisierung?

6. Um *gute* Modularisierung technisch überhaupt zu ermöglichen, muss das Modularisierungskonzept einer Sprache das *Geheimnisprinzip* erfüllen. Was ist damit gemeint?

7. Haskells Modulkonzept unterstützt keinen *automatischen Reexport*. Was ist damit gemeint?

8. Wie ist in Haskell vorzugehen, wenn ein Reexport nötig ist?

9. Was verstehen wir unter einem

   (a) selektiven

   (b) nicht selektiven

   Import, Export?

10. Auf welche Weisen sind in Haskell ein

    (a) selektiver

    (b) nicht selektiver

    Import, Export möglich?

11. Werden beim Export algebraischer und Neuer Typen in Haskell nur die Typnamen oder auch deren Datenkonstruktoren mit exportiert?

12. Wie ist ein Modul in Haskell grundsätzlich aufgebaut?

13. Wann liegt ein Namenskonflikt beim Import von Modulen vor?

14. Welche Möglichkeiten gibt es in Haskell mit Namenskonflikten beim Import von Modulen umzugehen?

15. Welche Konventionen sollen beim Schreiben von Modulen in Haskell eingehalten werden?

16. Was ist die Idee hinter dem Konzept abstrakter Datentypen?

17. Welche Herausforderungen sind bei der Definition eines abstrakten Datentyps zu meistern?

18. Warum benötigt man für die Implementierung eines abstrakten Datentyps doch (auch) einen konkreten Datentyp?

19. Inwiefern ist Haskells Modulkonzept nützlich für die Implementierung abstrakter Datentypen in Haskell?

20. Spezifizieren Sie nach dem Vorbild des abstrakten Datentyps Warteschlange mit *first-in/first-out* (FIFO) Verhalten den abstrakten Datentyp Stapel mit *last-in/first-out* (LIFO) Verhalten.

21. Implementieren Sie ebenfalls nach Vorbild des abstrakten Datentyps Warteschlange auch den abstrakten Datentyp Stapel mithilfe verschiedener konkreter Datentypen.

22. Wie ist die Implementierung abstrakter Datentypen in Haskell von einem programmiertechnischen Standpunkt aus zu beurteilen? Gibt es Schwächen?

23. Was wäre eine bessere Möglichkeit, wenn wir die Sprache Haskell erweitern, verändern dürften?

24. Auf wen gehen grundlegende Arbeiten zum Konzept abstrakter Datentypen zurück?

25. Was sind generelle Vorteile, Gewinne aus der Möglichkeit, Programme zu modularisieren?

Teil I – V, Verschiedene Kapitel

1. Lösen Sie die Aufgabe des Umkehrens von Zeichenreichen

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse x ++ [x]
```

   (a) repetitiv rekursiv.

   (b) ohne Verwendung des Operators `(++)`.

2. Welchen allgemeinsten Typ haben folgende Funktionen und woran erkennt man das?

   (a) `f x y = (x + y) * (x - y)`

   (b) `g k x y = k (x,y)`

   (c)
```
h [] = []
h (x:xs) = (h xs) ++ [x]
```

3. Welche der folgenden Ausdrücke sind gültig? Welche nicht? Was ist der Wert der gültigen Ausdrücke?

   (a) `fst (2,3)`

   (b) `fst (2,3,5)`

   (c) `fst [2,3]`

   (d) `fst [2,3,5]`

   (e) `head (2,3)`

   (f) `head (2,3,5)`

   (g) `head [2,3]`

   (h) `head [2,3,5]`

   (i) `tail (2,3)`

   (j) `tail (2,3,5)`

   (k) `tail [2,3]`

   (l) `tail [2,3,5]`

   (m) `last (2,3)`

   (n) `last (2,3,5)`

   (o) `last [2,3]`

   (p) `last [2,3,5]`

4. Illustrieren Sie die Aussage des *Hauptleitsatzes fkt. Programmierung* anhand geeigneter Beispiele.

5. Welche Performanzfallen lauern bei unbedachter Verwendung von Rekursion?

6. Geben Sie ein Beispiel an, wo es eine solche Falle gibt.

7. Wie lassen sich diese Fallen (oft) vermeiden?

8. Wenden Sie die Vermeidungsmethode auf Ihr Beispiel aus der vorausgehenden Aufgabe an.

9. Welche Berechnungskomplexität haben folgende Funktionen:

   (a) Fakultät

   ```
   fac :: Int -> Int
   fac n
    | n == 0 = 1
    | n >= 1  = n * fac (n-1)
   ```

   (b) Fibonacci

   ```
   fib :: Int -> Int
   fib n
    | n == 0 = 0
    | n == 1 = 1
    | n >= 2  = fib (n-1) + fib (n-2)
   ```

   (c) Bionimialkoeffizieten

   ```
   binom :: Int -> Int -> Int
   binom n k = div (fac n) (fac k * fac (n-k))
    | n == 0 = 1
    | n >= 1  = n * fac (n-1)
   ```

10. Einige der folgenden Auswertungsstrategien bedeuten dasselbe. Welche?

    (a) Normal
    (b) Applikativ
    (c) Linksnormal
    (d) Linksapplikativ
    (e) Rechtsnormal
    (f) Rechtsapplikativ
    (g) Linkestinnerst
    (h) Linkestäußerst
    (i) Rechtestinnerst
    (j) Rechtestäußerst
    (k) Faul
    (l) Fleißig

    sie für die Auswertung?

11. Zeigen Sie, dass die linke und rechte Seite der definierenden Gleichung von `flip` denselben Typ besitzen:

    ```
    flip :: (a -> b -> c) -> (b -> a -> c)
    flip f x y = f y x
    ```

12. Auf welche Weise können üblicherweise als Infixoperatoren gebrauchte Operatoren in Haskell als Präfixoperatoren gebraucht werden und umgekehrt? Illustrieren SIe Ihre Antwort an je einem Beispiel.

13. Welche Eigenschaft muss ein Operator haben, damit man in Haskell einen Operatorabschnitt bilden kann?

14. Listen sind in Haskell schon vordefiniert. Definieren Sie induktiv einen eigenen parametrisch polymorphen Listentyp, für den folgendes gilt:

    (i) `LL` bezeichnet die leere Liste.
    (ii) Ist `l` eine Liste und `w` ein Wert vom Listenelementtyp, so vereinigt `Cons` Liste `l` und Element `w` zu einer neuen Liste, in der `w` als letztes Element an `l` angefügt ist.

15. Für die Implementierung des Datentyps `Haustier` stehen folgende zwei Typvereinbarungen zur Wahl:

```
type Name = String
data Lieblingsplatz = Sofa | Fensterbank | Futternaph | Bett | FeldWaldundWiese deriving Eq
type HoertAuf = Zuruf | Manchmal | Garnicht deriving Eq

newtype Haustier = H (Name,Lieblingsplatz,HoertAuf)
data Haustier = H (NameLieblingsplatz,HoertAuf)
```

Was spricht für, was gegen die Vereinbarung von `Haustier` als

(a) Neuer Typ?

(b) algebraischer Typ?

16. Wer war Haskell B. Curry? Welche Leistungen sind mit ihm verbunden? Wofür ist er bekannt geworden?

17. *Referentielle Transparenz* ist lt. Steckbrief funktionaler Programmierung charakteristische Eigenschaft funktionaler Programmierung. Was ist mit diesem Begriff gemeint?

18. Was haben alle Funktionen, die durch λ-Lifting entstehen, gemeinsam?

19. Erklären Sie die Idee partieller Auswertung anhand von Operatorabschnitten.

20. Warum sind in Haskell zwei Funktionale zum Falten von Listen vordefiniert?

21. Angewendet auf die gleichen Argumente liefern die Funktionen `foldl`, `foldr` stets dasselbe Resultat. Richtig oder falsch? Begründen Sie Ihre Antwort.

22. Was besagt der *Hauptleitsatz funktionaler Programmierung*?

23. Funktionen höherer Ordnung und Polymorphie versprechen eine höhere Programmiereffizienz. Warum? Begründen Sie Ihre Antwort.

24. Was sind typische Fragen, die in der Berechnungstheorie gestellt und untersucht werden?

25. Was ist hier falsch? Markieren und verbessern Sie alle syntaktischen Fehler:

```
data (Tree (a b c)) = Leaf a b
           | root (Tree a
               b c)
           c
      Tree (a b c)

depth :: Tree -> Int
 depth (Leaf _)     = 0 :: Int
 depth (root l _ r) = max (depth l) (depth r) + 1

sumtree :: (Tree a b c) -> Int
sumtree (Leaf x y)  = x+y
sumtree root l _ r) = sumtree l ++ sumtree r

flatten :: a -> b -> c -> Tree (a b c) -> (c)
flatten f (Leaf x y)
  = f (x y)
flatten f (root l z r) = (flatten f l) ++ [z]
 ++ flatten (f r)
```

Überprüfen Sie mit ghc oder Hugs, ob Sie alle Fehler gefunden haben.