LVA 185.A03 Funktionale Programmierung (WS 20)

Leit- und Kontrollfragen IV

Mi, 04.11.2020

Stoff: Vorlesungsteil IV – Kapitel 10 und 11

Funktionale Programmierung – Funktionen höherer Ordnung, Polymorphie

(Ohne Abgabe, ohne Beurteilung; zur Selbsteinschätzung)

Teil IV, Kapitel 10 'Funktionen höherer Ordnung'

1. Woran erkennt man eine Funktion
   (a) erster
   (b) höherer

   Ordnung?
2. Welche kürzere Sprechweise gibt es für Funktionen höherer Ordnung?
3. Geben Sie einige Beispiele in Haskell vordefinierter Funktionen höherer Ordnung an auf
   (a) numerischen Typen
   (b) Listentypen
   (c) Wahrheitswerten
4. Welche Beispiele für Funktionen höherer Ordnung haben Sie bereits in
   (a) der Mathematik
   (b) anderen Bereichen der Informatik

   kennengelernt?
5. Was ist mit dem Schlagwort von Funktionen als *first class citizens* gemeint?
6. Was versteht man unter funktionaler Abstraktion?
7. Was ist das Pendant zu funktionaler Abstraktion in
   (a) prozeduralen
   (b) objektorientierten

   Programmiersprachen?
8. Was geschieht bei funktionaler Abstraktion
   (a) erster
   (b) höherer

   Stufe?
9. Illustrieren Sie funktionale Abstraktion
   (a) erster
   (b) höherer

   anhand je eines geeigneten Beispiels.
10. Wie spielen Funktionen höherer Ordnung und Curryfizierung einander in die Hände?
11. Funktionale Abstraktion unterstützt Wiederverwendung.

(a) In welcher Weise?

(b) Wovon?

12. Welche Art von Wiederverwendung bzw. von was unterstützt funktionale Abstraktion

    (a) erster

    (b) höherer

    Stufe?

13. Welche funktionalen Argument oder/und Resultate bzw. Argument oder/und Resultattypen haben die Funktionen `zip` und `zipWith`?

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

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f  _      _     = []
```

14. Implementieren Sie die Funktion `zip` mithilfe von `zipWith`.

15. Nennen Sie einige Beispiele in Haskell vordefinierter Funktionen mit funktionalen

    (a) Argumenten

    (b) Resultat.

    Geben Sie auch die Typsignatur ihrer Beispiele an und heben Sie die funktionalen Argumente und/oder Resultate durch Klammerung besonders hervor.

16. Was ist mit λ-*lifting* gemeint?

17. In welcher Weise spielen Curryfizierung und partielle Auswertung von Funktionen einander in die Hände?

18. Was ist Funktionskomposition, was Funktionsapplikation? Erklären Sie den Unterschied.

19. Woran erkennt man

    (a) Funktionskomposition

    (b) Funktionsapplikation

    in Haskell (syntaktisch)?

20. Wozu dienen die Funktionen

    – `foldl`

    – `foldr`?

    Wie bzw. worin unterscheiden sie sich?

21. Was berechnen bzw. welchen Wert haben folgende Ausdrücke?

    (a) `foldl (^) 2 [1,2,3]`

    (b) `foldr (^) 2 [1,2,3]`

22. Im Standard-Präludium ist die Funktion `concat` wie folgt definiert:

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

    (a) Wozu dient `concat`, welche Bedeutung hat es?

(b) Wäre eine Implementierung von `concat` mit `foldl` anstelle von `foldr` genauso möglich? Wäre es sinnvoll? Begründen Sie Ihre Antwort.

23. Was ist mit applikativem Rechnen gemeint? Was mit funktionalem?

24. Erläutern Sie den Unterschied applikativen und funktionalen Rechnens anhand des Algorithmus von Euklid.

25. Übertragen Sie die Idee applikativen und funktionalen Rechnens im Algorithmus von Euklid auf die Berechnung des kleinsten gemeinsamen Vielfachen zweier natürlicher Zahlen m und n mit $m, n \geq 1$.

Teil IV, Kapitel 11 'Polymorphie'

1. Gegeben sind die Funktionen `change` und `mapfun`:

```
change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g = \z -> if z == x then y else f z

mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f []           = f
mapfun f ((x,y) : zs) = mapfun (change f x y) zs
```

   (a) Warum muss bei `change` eine Kontexteinschränkung für die Typvariable `a` vorgenommen werden, nicht aber für `b`?

   (b) Warum gilt das in gleicher Weise für `mapfun`?

2. Was ist die Bedeutung der Funktion `change`, was von `mapfun`? Erklären Sie knapp, aber gut nachvollziehbar, wie die beiden Funktionen arbeiten.

3. Sind die Funktionen `change` und `mapfun` curryfiziert oder uncurryfiziert?

4. Was ist die

   (a) cuuryfizierte

   (b) uncurryfizierte

   Lesart von `change` und `mapfun`?

5. Geben Sie einige Beispiele in Haskell vordefinierter

   (a) parametrisch

   (b) *ad hoc*

   polymorpher Funktionen an.

6. Von welchem Polymorphietyp sind folgende Funktionen aus Standard-Präludium bzw. Kapitel 10:

   (a) `(+) :: Num a => a -> a -> a`

   (b) `rekSchema :: (Num a, Ord a) => b -> (a -> b -> b) -> a -> b`

   (c) `flip :: (a -> b -> c) -> (b -> a -> c)`

   Geben Sie den Polymorphietyp (und seine Synonyme) möglichst genau an.

7. Ist

```
f :: a -> b
```

   sinnvoll als Typdeklaration einer polymorphen Funktion? Begründen Sie Ihre Antwort. Überlegen Sie, ob sich die Deklarationszeile zu einer vollständigen Implementierung ergänzen lässt.

8. Ändert sich Ihre Antwort, wenn wir statt `f` folgende Funktion `g`:

   ```
   g :: a -> a
   ```

   betrachten? Wenn ja, wie?

9. Ergänzen Sie (möglichst allgemein) den Typ der Funktionskomposition:

   ```
   (.) :: ...
   (.) f g = \x -> f (g x)
   ```

10. Was ist Überladung?

11. Wodurch unterscheidet sich direkte und indirekte Überladung?

12. Was erlaubt Polymorphie auf Datentypen wiederzuwenden?

13. Was unterscheidet Überladung und echte Polymorphie?

14. Insert 'some' and 'any' appropriately: A parametric polymorphic function works for ... data type, an *ad hoc* polymorphic function works for ... data type.

15. Gegeben ist folgende Funktion:

    ```
    f []         = ([],[])
    f ((x,y):zs) = (x:xs,y:ys) where (xs,ys) = f zs
    ```

    (a) Welches ist der allgemeinste Typ von `f`?

    (b) Von welchem Polymorphietyp ist `f`, wenn es polymorph ist?

    (c) Was 'macht' `f`? Beschreiben Sie knapp, aber gut nachvollziehbar, was `f` 'macht', welche Bedeutung es hat.

16. Was erlaubt

    (a) echte

    (b) unechte

    Polymorphie auf Funktionen wiederzuverwenden?

17. Woran erkennt man eine

    (a) echte

    (b) unecht

    polymorphe Funktion?

18. Unechte Polymorphie auf Funktionen lässt sich in direkte und indirekte unechte Polymorphie scheiden; echte Polymorphie nicht. Richtig oder falsch?

19. Wie spielen Typklassen und unechte Polymorphie einander in die Hände?

20. Was ist damit gemeint, dass Typklassen voneinander erben können?

21. Typklassen in Haskell dienen der Organisation von Überladung. Wie ist das zu verstehen? Was ist damit gemeint?

22. Überladung von Funktionen für neue Typen lässt sich manchmal automatisch vornehmen.

    (a) Welche Funktionen kommen dafür (ausschließlich) infrage?

    (b) Wo liegen die Grenzen der automatischen Überladung dieser Funktionen?

    (c) Wie kann die Überladung erreicht werden, wenn der Automatismus dafür nicht (mehr) infrage kommt?

    (d) Welche Haskell-Sprachkonstrukte dienen zur automatischen bzw. nichtautomatischen Überladung?

(e) Gibt es generelle Grenzen von Überladung, d.h. gibt es Typen, fúr die eine grundsätzlich überladungsfähige Funktion nicht überladen werden kann? Begründen Sie Ihre Antwort.

23. Für echt polymorphe Funktionen gilt: Ein Funktionsname, eine Implementierung. Was gilt entsprechend für unecht polymorphe Funktionen?

24. Polymorphie verspricht höhere

    (a) Transparenz, Lesbarkeit

    (b) Verlässlichkeit, Wartbarkeit

    von Programmen. Warum? Begründen Sie Ihre Antwort.

25. Wie wirkt Polymorphie der Knappheit guter Namen entgegen?

Teil I – III, Verschiedene Kapitel

1. Wie unterscheiden sich die Typen `Float` und `Double` voneinander?

2. Was ist ein Relator? Was ist eine Relation?

3. Was sind Listenkomprehensionsausdrücke? Illustrieren Sie Ihre Antwort mit einigen Beispielen.

4. Woran erkennt man:

    (a) schlicht

    (b) linear

    (c) baumartig

    (d) geschachtelt

    rekursive Funktionen?

5. Welche synonyme Bezeichnungen gibt es für die Rekursionsarten aus der vorigen Teilaufgabe?

6. Geben Sie ein Beispiel an für zwei (oder mehr) wechselweise rekursive Funktionen.

7. Welche Begriffe sind statt wechselweiser Rekursion auch üblich?

8. Funktionen können in Haskell mithilfe verschiedener Sprachmittel definiert werden.

    (a) Nennen Sie möglichst viele.

    (b) Welches Sprachmittel werden dabei am häufigsten benutzt? Warum?

9. Welches Symbol erlaubt es, mehr als eine Gleichung in eine Zeile zu schreiben?

10. Welche Klammererung gilt in Haskell implizit für

    (a) Funktionssignaturen

    (b) Funktionsterme?

11. Was versteht man unter curryifzierten Funktionen? Was unter uncurryfizierten? Illustrieren Sie Ihre Antwort anhand geeigner Beispiele.

12. Geben Sie die Signatur und Implementierung der vordefinierten Funktionen `curry` und `uncurry` an.

13. Gegeben ist die Deklaration:

```
foldFrom :: (a -> b -> a) -> a -> [b] -> (a -> Bool) -> a
foldFrom f e [] _    = e
foldFrom f e xxs@(x:xs) p
 | p a  = foldl f e xxs
 | True = foldFrom f e xs p
```

Was ist die Bedeutung von `foldFrom`? Erklären Sie knapp, aber gut nachvollziehbar, was sich aus den einzelnen Zeilen der Deklaration von `foldFrom` herauslesen lässt und wie die Auswertung der Funktion vorgeht.

14. Sind

    (a) `xs @ (x : (y : (z : zs)))`

    (b) `xs @ (ys ++ zs)` gültige Muster(ausdrücke)? Begründen SIe Ihre Antwort.

15. Was ist mit gemeint, wenn man vom *von Neumann*-Stil spricht?

16. Wofür können in Haskell 'sprechende' Namen selbst gewählt werden?

17. Was ist hier falsch? Markieren und verbessern Sie alle Fehler, so dass sich das Programm interpretieren und übersetzen lässt:

```
typ Verwandt, Bekannt = Bool

data Wohnform :: EFH, ZFH, MFH, DH, RH, HH, PH Deriving Eq Show

newtype Bürger   = B Person Wohnform Nachbarn Freunde deriving Eq | Show
newtype Nachbarn = N [(Bürger,Verwandt,Bekannt)] deriving Eq | Show
newtype Freunde  = F [Bürger] deriving Eq | Show

verwandteNachbarn : Bürger -> [Person]
verwandteNachbarn (B - - (N ls) -)
 = [p : (B p _ _ _;verwandt;_) <- ls; verwandt = true]

verwandteNachbarnNamen :: Bürger => ([Nachname,Vorname])
verwandteNachbarnNamen (B _ _ (N ls) _)
 = [(nn,vn) || (B (P vn nn _) _ _ _ vw _) <= ls && not vw <> true]
```

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

18. Definieren Sie einen Datentyp, der es erlaubt, für verschiedene Modelle von Autoherstellern verschiedener Marken wesentliche Eigenschaften wie Antriebsart, Abmessungen, Motorleistung und Listenpreis abzulegen.

19. Geben Sie möglichst genau an, was für einen Datentyp Sie in der vorigen Aufgabe gewählt haben. Geben Sie auch an, warum Sie den Datentyp so und nicht anders gewählt haben.

20. Machen Sie Ihren Datentyp (wenn möglich) aus der vorvorgehenden Aufgabe zu Instanzen der Typklassen

    (a) `Eq`

    (b) `Show`

    mithilfe expliziter `instance`-Deklarationen.

21. Falls Sie Ihren Datentyp aus der Voraufgabe nicht zu einer Instanz der von `Eq` und `Show` machen können, was ist der Grund dafür? Was müssten Sie ändern, um die Instanzbildungen vornehmen zu können?

22. Implementieren Sie die Funktion *zwei-hoch*, in Zeichen $2^{\cdot}$:

$$2^{\cdot} : \mathbb{N} \to \mathbb{N}$$

$$\forall\, n \in \mathbb{N}.\ 2^n = \begin{cases} 1 & \text{falls } n = 0 \\ 2 * 2^{n-1} & \text{falls } n > 0 \end{cases}$$

    in möglichst vielen syntaktischen Varianten in Haskell (mit Fallunterscheidungen, bewachten Ausdrücken, Mustern, Listenkomprehensionen, anonymen λ-Abstraktionen,...).

23. Typklassen wie `Ord` sehen vor, dass für ihre Instanzen sehr viele Funktionen implementiert sein müssen. Bei einer konkreten Instanzbildung sind tatsächlich aber oft nur sehr wenige zu implementieren. Warum?

24. Welche Funktionen reichen oft aus, bei Instanzbildungen für die Typklasse `Ord` implementiert zu werden?

25. Wann ist es nötig, mehr als die Funktionen aus der Voraufgabe zu implementieren? Illustrieren Sie Ihre Antwort mit einem passenden Beispiel.