LVA 185.A03 Funktionale Programmierung (WS 20)

Leit- und Kontrollfragen VII

Mi, 16.12.2020

Stoff: Vorlesungsteil VI und VII – Kapitel 18 und 19

Weiterführende Konzepte und Abschluss – Programmierprinzipien, Rückschau/Ausschau

(Ohne Abgabe, ohne Beurteilung; zur Selbsteinschätzung)

Teil VI, Kapitel 18 'Funktionale und allgemeine Programmierprinzipien'

1. Was ist die Idee *reflexiven Programmierens* nach Simon Thompson?
2. Beschreiben Sie knapp, aber prägnant die verschiedenen Phasen reflexiver Programmierung.
3. Welche Schlüssel- und Leitfragen sind für diese Phasen typisch? Nennen Sie je einige Beispiele.
4. In welche Phase gehört die Frage "ob sich das Problem verallgemeinern und so möglicherweise einfacher lösen lässt?"
5. Vergleichen Sie die Idee reflexiven Programmierens mit den Entwurfsprozessen nach
   (a) Graham Hutton
   (b) Norman Ramsey

   aus Kapitel 9. Welche Gemeinsamkeiten, Unterschiede fallen Ihnen auf?
6. Was sind *nichtfunktionale* Eigenschaften eines Programms im Unterschied zu *funktionalen*? Nennen Sie einige Beispiele typischer nichtfunktionaler Eigenschaften von Programmen.
7. In welche Phase gehört die Frage "ob das Programm auch nichtfunktionale Eigenschaft gut erfüllt"?
8. Was bedeutet es, dass ein Problem
   (a) überspezifiziert
   (b) unterspezifiziert

   ist? Warum muss beides vermieden werden?
9. In welche Phase gehört die Frage "ob das Problem über- oder unterspezifiziert ist"?
10. Warum ist *reflexives Programmieren* ein allgemeines Programmierprinzip, nicht nur ein funktionales?
11. Was ist mit *Kapseln algorithmischen Vorgehens* gemeint?
12. Illustrieren Sie die Idee anhand eines Beispiels.
13. Welches Sprachmittel funktionaler Sprachen ist für die Kapselung entscheidend?
14. Warum ist das Teile-und-Herrsche als Berechnungsverfahren für die Berechnung der Fibonacci-Zahlen nicht geeignet?
15. Wie geht *mergeSort* zum Sortieren von Listen vor? Implementieren Sie das Verfahren in Haskell. Lassen Sie sich ggf. von der Darstellung auf `http://idea-instructions.com/merge-sort` leiten.
16. Wenden Sie das Vorgehen von Teile-und-Herrsche nach dem Vorbild von *quickSort* auf *mergeSort* an.
17. Wie unterscheiden sich die Implementierungen aus den vorigen beiden Teilaufgaben voneinander? Welche scheint Ihnen konzeptuell und implementierungstechnisch einfacher zu sein? Warum?

18. Welche anderen Probleme haben Sie schon kennengelernt, die sich mittels Teile-und-Herrsche gut lösen lassen? Wo haben Sie diese Probleme kennengelernt?

19. Lösen Sie einige dieser Probleme mithilfe des gekapselten Teile-und-Herrsche-Verfahrens aus Kapitel 18.

20. Wenn Sie einige dieser Probleme in anderen Lehrveranstaltungen schon gelöst haben, wie unterscheiden sich diese Lösungen von ihren Lösungen aus der vorigen Aufgabe?

21. Warum ist die beschriebene Idee zur Kapselung ein funktionales Programmierprinzip, aber kein allgemeines? Z.B. kein objektorientiertes?

22. Was versteht man unter einem *Strom*, was unter *Stromprogrammierung*?

23. Was sind *generiere/anpass*-Modularisierungen?

24. Welche Eigenschaft funktionaler Sprachen ist dafür entscheidend? Warum?

25. Illustrieren Sie am Beispiel des *Siebs des Eratosthenes* eine *generiere/selektiere*-Modularisierung.

26. Was unterscheidet *generiere/anpass*-Modularisierungen konzeptuell von 'gewöhnlichen' Modularisierungen?

27. Zeigen Sie anhand von Beispielen wie ein

    (a) *generiere*-Modul mit vielen *selektiere*-Moduln

    (b) *selektiere*-Modul mit vielen *generiere*-Moduln

    kombiniert werden kann.

28. Ist der Unterschied zwischen einem *selektiere*- und einem *filtere*-Modul klar definiert? Wie könnte der Unterschied pragmatisch definiert werden?

29. Illustrieren Sie die Idee der *generiere/selektiere*-Modularisierung graphisch.

30. Warum ist Terminierung für *generiere/anpass*-Modularisierungen auch in Sprachen wie Haskell mit später Auswertung eine relevante Frage? Illustrieren Sie Ihre Antwort anhand eines Beispiels.

31. Geben Sie ein Beispiel für eine *generiere/ransformiere*-Modularisierung an.

32. Was ist informell mit dem *Münchhausen*-Prinzip gemeint? Illustrieren Sie die Idee anhand eines Beispiels.

33. Illustrieren Sie die Idee der *generiere/transformiere*-Modularisierung graphisch.

34. Geben Sie einige Beispiele für Modularisierungen an, wo mehrere Generatoren, Selektoren, Filter, Transformatoren zum Einsatz kommen.

35. Keiner der drei Schritte des *Siebs des Eratosthenes* terminiert:

    (i) Schreibe *alle* (unendlich vielen!) natürlichen Zahlen ab 2 hintereinander auf.

    (ii) Die kleinste nicht gestrichene Zahl ist eine Primzahl. Streiche *alle* (unendlich vielen!) Vielfachen dieser Zahl.

    (iii) Wiederhole den vorigen Schritt (unendlich oft!) mit der jeweils nächstkleinsten noch nicht gestrichenen Zahl.

    Dennoch kann aufbauend auf dieser Idee ein effektives Haskell-Programm zur Berechnung z.B. der ersten 1.000 Primzahlen geschrieben werden. Warum? Ist das in gleicher Weise in jeder funktionalen Programmsprache möglich? Begründen Sie Ihre Antwort.

36. Warum sind *generiere/anpass*-Modularisierungen ein funktionales Programmierprinzip, aber kein allgemeines? Z.B. kein objektorientiertes?

37. Geben Sie einen Haskell-Ausdruck an, dessen Wert der Strom der Potenzen von 2 ist.

38. Passen Sie den Haskell-Ausdruck aus der vorigen Aufgabe so an, dass sein Wert die 2er Potenzen zwichen 1.000 und 100.000 sind.

39. Haben Sie die vorigen Aufgaben mit einer Listenkomprehension gelöst? Wenn nein, probieren Sie es mit einer Listenkomprehension. Wenn ja, probieren Sie es ohne.

40. Welche der Lösungen aus den beiden Voraufgaben erscheint Ihnen angemessener? Warum?

Teil VII, Kapitel 19 'Rückschau, Ausschau'

1. Durch welche Charakteristika und Eigenschaften unterscheiden sich

   (i) funktionale

   (ii) imperative

   Programmierung voneinander?

2. Peter Pepper schreibt in seinem Buch über funktionale Programmierung in Opal, ML, Haskell und Gofer: "Die Fülle an Möglichkeiten [in fkt. Sprachen] erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien". An welche Prinzipien denkt Pepper dabei? Im Zshg. mit

   (a) Funktionen?

   (b) Datenstrukturen?

3. Funktionen sind *erstrangige* Sprachelemente in funktionalen Sprachen. Was ist damit gemeint?

4. Konrad Hinsen verspricht in seinem Artikel über *The Promises of Functional Programming* die Versprechen funktionaler Programmierung aufzudecken. Welche sind das? Welche deckt er auf?

5. Welche Versprechen überlässt Hinsens Artikel der darauf aufbauenden Arbeit von Konstantin Läufer und George Thiruvathukal?

6. Wie würden Sie die Frage von John W. Backus aus seiner Rede aus Anlass der Verleihung des *Turing Award* im Jahr 1977 beantworten, ob *"die Programmierung vom von Neumann-Stil befreit werden könne?"* Begründen Sie Ihre Antwort.

7. In welchen weiteren regelmäßig angebotenen Lehrveranstaltungen an der TU Wien stehen Fragen funktionaler Programmierung im Vordergrund?

8. Welches sind die großen Themenkreise dieser Veranstaltungen?

9. Nach Edsger W. Dijkstra, *Turing Award* Preisträger von 1972, ist eine hervorragende Zukunft funktionaler Programmierung höchstens durch eine Sache behindert. Welche? Stimmen Sie ihm zu? Begründen Sie Ihre Antwort.

10. Was ist der *Turing Award*? Wer verleiht ihn? Wie oft? Wofür? Womit wird der Preis oft verglichen und gleichgesetzt?

11. Wurde jemandem der *Turing Award* für Beiträge zur oder mit Bezug zur funktionalen Programmierung verliehen?

12. Peter Pepper schreibt in *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, Springer-V., 2. Auflage, 2003: "*Functional programming is fun*, weil funktionale Programmierung etwas von der Eleganz der Mathematik in die Programmierung bringt." Stimmen Sie zu? Begründen Sie Ihre Antwort.

13. Mihai Maruseac schreibt in *Haskell: A Language for Modern Times*, Crossroads, the ACM Magazine for Students 24(1):64-66, 2017: "Learning Haskell opens one's mind to new programming paradigms, which might produce clearer and shorter implementations." Stimmen Sie zu? Begründen Sie Ihre Antwort wieder.

14. Yaron Minsky schreibt in *OCaml for the Masses*, Communications of the ACM 54(11):53-58, 2011: "[...] why the next language you learn should be functional!" Stimmen Sie zu? Begründen Sie Ihre Antwort auch hier.

15. Sehen Sie die Versprechen funktionaler Programmierung gegeben von Konrad Hinsen in *The Promises of Functional Programming*, Computing in Science and Engineering 11(4): 86-90, 2009, und Konstantin Läufer und George K. Thiruvathukal in *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*, Computing in Science and Engineering 11(5): 68-75, 2009, als erfüllt an? Begründen Sie Ihre Antwort.

Teil I – VI, Verschiedene Kapitel

1. Was ist mit *Rechnen mit Funktionen* gemeint? Ilustrieren Sie Ihre Antwort auch anhand einiger aussagekräftiger Beispiele.
2. Was unterscheidet applikative von funktionaler Programmierung?
3. Ist applikative Programmierung im engeren Sinn in einer vollausgebildeten fkt. Sprache wie Haskell überhaupt möglich? Begründen Sie Ihre Antwort.
4. Betrachten Sie die beiden Auswertungen des Terms `3 * 5 + square (2 + 3) + 5 - 3`

```
3 * 5 + square (2 + 3) + 5 - 3
  ->> 15 + square (2 + 3) + 5 - 3
  ->> 15 + (2 + 3) * (2 + 3) + 5 - 3
  ->> 15 + 5 * (2 + 3) + 5 - 3
  ->> 15 + 5 * 5 + 5 - 3
  ->> 15 + 25 + 5 - 3
  ->> 40 + 5 - 3
  ->> 45 - 3
  ->> 42

3 * 5 + square (2 + 3) + 5 - 3
  ->> 3 * 5 + square (2 + 3) + 2
  ->> 3 * 5 + square 5 + 2
  ->> 3 * 5 + 5 * 5 + 2
  ->> 3 * 5 + 25 + 2
  ->> 3 * 5 + 27
  ->> 15 + 27
  ->> 42
```

   über der Funktion

```
square :: Int -> Int
square n  = n * n
```

   (a) Welche Auswertungsstrategie ist jeweils angewendet worden?
   (b) Woran erkennt man das jeweils?

5. Gegeben ist der Term

```
(2 + 3) * 5 + (fac (5-3) + 3) * 7 + (-5) + 1
```

   über der Fakultätsfunktion `fac`:

```
fac :: Nat0 -> Nat1
fac n = if n == 0 then 1 else n * fac (n-1)
```

   Wählen Sie einige der nachstehenden Strategien und werten Sie den Term entsprechend der gewählten Strategien aus:

   (i) Normal
   (ii) Linksnormal

(iii) Rechtsnormal
(iv) Applikativ
(v) Linksapplikativ
(vi) Rechtsapplikativ

6. Sind

   (i) Mittignormal
   (ii) Mittigapplikativ

   auch sinnvolle Auswertungsstrategien? Begründen Sie Ihre Antwort.

7. *Rechnen auf Parameterposition* erlaubt lineare Rekursion durch einen anderen Rekursionstyp zu ersetzen. Welchen? Warum ist diese Ersetzung wichtig?

8. Auf welchen Säulen ruhen Stärke und Eleganz fkt. Programmierung?

9. Wie passt das Aristoteles-Zitat *"Das Ganze ist mehr als die Summe seiner Teile"* auf funktionale Programmierung?

10. Was legen folgende Auswertungsstrategien fest? Welche Freiheitsgrade lassen sie für die Auswertung?

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

11. Gegeben sind folgende Listentypen:

```
data Liste a  = LL  | Cons (Liste a) a
data Liste' a = LL' | Cons' a (Liste' a)
```

    Schreiben Sie zwei Haskell-Rechenvorschriften:

    (a) `L2L'`
    (b) `L'2L`

    die Werte vom Typ `Liste` in ihre entsprechenden `Liste'`-Werte überführt und umgekehrt. Geben Sie dabei auch die vollständigen Typsignaturen für `L2L'` und `L'2L` an. Nehmen Sie Typklasseninstanzbildungen für `Liste` und `Liste'` vor, wenn für `L2L'` und `L'2L` nötig; aber auch nur dann.

12. Sind die Funktionen `L2L'` und `L'2L` monomorph oder polymorph? Falls polymorph, welche Art von Polymorphie liegt vor? Welche anderen Bezeichnungen sind für diesen Polymorphiebegriff üblich?

13. Was ist der allgemeinste Typ von `f`?

```
f g p [] = []
f g p (x:xs)
 | p x       = g x : f g p xs
 | otherwise = []
```

14. Welche Eigenschaft beschreibt den

    (a) innneren

    (b) äußeren

    Zusammenhang von Modulen?

15. Welche Berechenbarkeitsmodelle können Sie nennen? Mit welchen davon haben Sie sich in dieser oder anderen Lehrveranstaltungen schon genauer beschäftigt?

16. Warum sind angewandte λ-Kalküle in der Berechenbarkeitstheorie von geringer Bedeutung?

17. Warum ist die α-Konversion im λ-Kalkül unverzichtbar?

18. Warum stellen die Church-/Rosser-Theoreme die Grundlage für die Definition der Semantik des reinen λ-Kalküls dar?

19. Worin liegt die Bedeutung des λ-Kalküls für die funktionale Programmierung? Worin für Haskell?

20. Warum ist die linksnormale Auswertungsordnung ohne implementierungspraktische Bedeutung?

21. Wann werden Typfehler in Programmen einer

    (a) stark

    (b) schwach

    getypten Sprachen aufgedeckt?

22. Haskell ist eine starkt getypte Sprache mit automatischer Typinferenz. Was bedeutet das?

23. Woran erkennt man in einem Typsystem wie in Kapitel 14.5 ein Axiom, woran eine Regel?

24. In prozeduralen Sprachen bereitet Ein-/Ausgabe konzeptuell keine Probleme, in funktionalen hingegen schon. Warum?

25. Von welchem Rekursionstyp ist die Ackermann-Funktion?