LVA 185.A03 Funktionale Programmierung (WS 20)

Leit- und Kontrollfragen V

Mi, 18.11.2020

Stoff: Vorlesungsteil V – Kapitel 12, 13 und 14

Fundierung funktionaler Programmierung – λ-Kalkül, Auswertungsordnungen, Typprüfung/Typinferenz

(Ohne Abgabe, ohne Beurteilung; zur Selbsteinschätzung)

Teil V, Kapitel 12 'λ-Kalkül'

1. Wofür ist der λ-Kalkül erdacht worden? Auf wen geht er zurück?
2. Was bedeutet *intuitiv berechenbar*, *intuitive Berechenbarkeit*?
3. Was unterscheidet intuitive von formaler Berechenbarkeit?
4. Was besagt die Churchsche These?
5. Ist die Churchsche These verifizierbar? Ist sie falsifizierbar? Begründen Sie Ihre Antwort.
6. Welche Bedeutung hat Wilhelm Ackermann für die Berechenbarkeitstheorie?
7. Wird *intuitive Berechenbarkeit* infrage gestellt? Wenn ja, warum und in welcher Weise?
8. Wodurch zeichnet sich der λ-Kalkül gegenüber anderen Berechnungsmodellen aus?
9. Was ist mit dem Bonmot der λ-Kalkül sei die Assembler-Sprache funktionaler Programmierung gemeint?
10. Was unterscheidet den reinen von sog. angewandten λ-Kalkülen?
11. Wie ist der reine λ-Kalkül syntaktisch aufgebaut? Welche Konstrukte gibt es?
12. Welche Regeln, Konventionen gelten im reinen λ-Kalkül zur Einsparung von Klammern?
13. Markieren Sie in den folgenden λ-Ausdrücken alle freien und gebundenen Vorkommen von Namen, bei gebundenen Vorkommen zusätzlich, woran sie gebunden sind:
    (a) $\lambda f.\ \lambda g.\ \lambda h.\ f\ (g\ h))\ \lambda g.\ ((\lambda g.\ g)\ g))$
    (b) $((\lambda x.\ \lambda y.\ x\ y)\ (((\lambda x.\ \lambda\ y.x\ y)\ a)\ b))\ c$
14. Markieren Sie in den folgenden λ-Ausdrücken die Bindungs- und Gültigkeitsbereiche aller definierenden Vorkommen von Namen:
    (a) $\lambda f.\ \lambda g.\ \lambda h.\ f\ (g\ h))\ \lambda g.\ ((\lambda g.\ g)\ g))$
    (b) $((\lambda x.\ \lambda y.\ x\ y)\ (((\lambda x.\ \lambda\ y.x\ y)\ a)\ b))\ c$
15. Was sind Ratoren in λ-Ausdrücken? Was Randen?
16. Was unterscheidet Konversions- und Reduktionsregeln des reinen λ-Kalküls?
17. Was bezeichnet man mit *syntaktischer Substitution*?
18. Warum kann syntaktische Substitution naiv angewendet zu Bindungsfehlern führen?
19. Wie vermeidet syntaktische Substitution bei korrekter, nicht naiver Anwendung Bindungsfehler?
20. Was versteht man unter einem Reduktionsschritt? Unter einer Reduktionsfolge?
21. Woran erkennt man, dass ein λ-Ausdruck in Normalform vorliegt?

22. Welcher Zusammenhang besteht zwischen den Begriffen normale, applikative, innerste, äußerste Reduktionsordnung?

23. Was legen die Begriffe normale, applikative, innerste, äußerste Reduktionsordnung für die Vereinfachung von λ-Ausdrücken fest? Was lassen sie offen?

24. Welche Reduktionsordnungen legen fest, was die Reduktionsordnungen aus der vorigen Aufgabe noch offen lassen? In welcher Weise treffen sie diese Festlegung(en)?

25. Welche Beinamen haben die sog. Church/Rosser-Theoreme? Was besagen sie?

26. Wie ist die Semantik von λ-Ausdrücken definiert?

27. Kann die Semantik eines λ-Ausdrucks undefiniert sein? Begründen Sie Ihre Antwort.

28. Woran erkennt man einen Kombinator im λ-Kalkül?

29. Welche Bedeutung hat der Y-Kombinator im λ-Kalkül?

30. Reduzieren Sie den λ-Ausdruck

    $((\lambda\, x.\ \lambda\, y.\ x\ y)\ (((\lambda\, x.\ \lambda\ y.x\ y)\ a)\ b))\ c$

    (a) normal

    (b) applikativ

    zur Normalform.

Teil V, Kapitel 13 'Auswertungsordnungen'

1. Was wird durch eine Auswertungsordnung festgelegt?

2. *Rechtsnormale* und *rechtsapplikative Auswertung* sind die dualen Gegenstücke *linksnormaler* und *linksapplikativer Auswertung.* Beschreiben Sie das Vorgehen

    (a) rechtsnormaler

    (b) rechtsapplikativer

    Auswertung.

3. Werten Sie den Term:

    ```
    2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
    ```

    (a) rechtsnormal

    (b) rechtsapplikativ

    aus.

4. Warum (vermutlich) sind rechtsnormale und rechtsapplikative Auswertung in (vermutlich) keinem Übersetzer und Interpretierer für eine funktionale Sprache implementiert zu finden?

5. Was bedeutet

    (a) normale

    (b) applikative

    Funktionstermauswertung?

6. In Haskell werden Funktionsterme standardmäßig normal ausgewertet. Richtig? Falsch? Ja, aber? Begründen Sie Ihre Antwort.

7. In imperativen Sprachen spricht man von *call by value*-Auswertung; in funktionalen Sprachen von ... Auswertung. Vervollständigen Sie den Satz entsprechend.

8. In funktionalen Sprachen spricht man von normaler Auwertung; in imperativen Sprachen von ... Auswertung. Vervollständigen Sie den Satz entsprechend.

9. In imperativen Sprachen spricht man von *call by need*-Auswertung; in funktionalen Sprachen von ... Auswertung. Vervollständigen Sie den Satz entsprechend.

10. Linksnormale und linksapplikative Auswertung liefern stets dasselbe Ergebnis. Richtig? Falsch? Ja, aber? Begründen Sie Ihre Antwort.

11. Was unterscheidet das Vorgehen von

    (a) applikativer, normaler

    (b) früher, später

    Auswertung?

12. Wie oft wird ein Argument eines Funktionsterms bei

    (a) applikativer

    (b) normaler

    (c) früher

    (d) später

    Auswertung ausgewertet?

13. Warum spricht man statt von applikativer auch von strikter Auswertung?

14. Für Haskell ist die späte Auswertungsordnung für Ausdrücke festgelegt. Etwas anderes ist nicht möglich. Richtig oder falsch? Begründen Sie Ihre Antwort.

15. Etwas salopp kann man sagen, dass die normale Auswertungsordnung am häufigsten terminiert. Was ist damit gemeint?

16. Frühe und applikative Auswertungsordnung bezeichnen dasselbe; späte und normale Auswertungsordnung ebenso. Richtig oder falsch? Begründen Sie Ihre Antwort.

17. Innerste und applikative Auswertungsordnung bezeichnen dasselbe; äußerste und normale Auswertungsordnung ebenso. Richtig oder falsch? Begründen Sie Ihre Antwort.

18. In keiner Auswertungsordnung werden Funktionstermargumente seltener ausgewertet als bei später Auswertungsordnung. Was heißt das genau und wie wird es technisch erreicht und sichergestellt?

19. Warum hat man sich in einigen funktionalen Sprachen für frühe, in anderen für späte Auswertung entschieden? Was spricht für frühe, was für späte Auswertung? Was spricht umgekehrt jeweils dagegen?

20. Was versucht eine Striktheitsanalyse über ein Programm herauszufinden?

Teil V, Kapitel 14 'Typprüfung, Typinferenz'

1. Wann spricht man von

    (a) stark

    (b) schwach

    getypten Sprachen?

2. Was spricht für die Wahl einer

    (a) stark

    (b) schwach

    getypten Programmiersprache?

3. Was unterscheidet Typprüfung und Typinferenz?
4. Was unterscheidet monomorphe und polymorphe Typen? Wofür stehen, was repräsentieren sie?
5. Wie kann das grundsätzliche Vorgehen bei Typprüfung und Typinferenz zusammengefasst und beschrieben werden?
6. Was bedeutet es für einen Ausdruck wohlgetypt zu sein?
7. Welche Rolle spielt Unifikation für polymorphe Typprüfung und Typinferenz?
8. Für monomorphe Typprüfung und Typinferenz wird Unifikation nicht gebraucht. Richtig oder falsch? Begründen Sie Ihre Antwort.
9. Inferieren Sie händisch den allgemeinsten Typ der Funktion `foldl`:

```
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

10. Was sind Typsprachen? Wozu dienen sie?
11. Was sind Typsysteme? Wozu dienen sie?
12. Warum kann man bei erfolgreicher Typprüfung, Typinferenz für ein Programm von einem teilweisen Korrektheitsbeweis für dieses Programm sprechen?
13. Was versteht man unter
    (a) der Instanz eines Typausdrucks
    (b) einer gemeinsamen Instanz einer Menge von Typausdrücken
    (c) der allgemeinsten Instanz einer Menge von Typausdrücken?
14. Warum ist der Ausdruck:

```
length ([]++[True]) + length ([]++[1,2,3])
```

    in Haskell wohlgetypt, der Ausdruck:

```
length (xs++[True]) + length (xs++[1,2,3])
```

    aber nicht?
15. Was bedeutet das Axiom VAR in Worten? Was die Regel COND?

$$\text{VAR} \quad \frac{}{\Gamma \vdash var : \Gamma(var)}$$

$$\text{COND} \quad \frac{\Gamma \vdash exp : Bool \quad \Gamma \vdash exp_1 : \tau \quad \Gamma \vdash exp_2 : \tau}{\Gamma \vdash \text{ if } exp \text{ then } exp1 \text{ else } exp2 : \tau}$$

Teil I – IV, Verschiedene Kapitel

1. Operatoren und Relatoren sind syntaktische Begriffe; Operationen und Relationen semantische. Erklären Sie den Unterschied anhand vordefinierter Operatoren und Relatoren in Haskell.
2. Geben Sie je einige gültige Werte folgender Typen an:
    (a) `[Int -> Int]`
    (b) `[Float -> Float]`
    (c) `[Char -> Int]`
    (d) `[Int -> IChar`
    (e) `[String -> String]`
    (f) `[String -> Char]`

(g) `[String -> Int]`

(h) `[Num a => a -> a -> a]`

(i) `[Ord a => a -> a -> Bool]`

3. Welchen allgemeinsten Typ haben folgende Ausdrücke und Teilausdrücke und woran erkennt man das jeweils?

   (a) `[] ++ [True]`

   (b) `x ++ y]`

   (c) `x : [y,'z']`

   (d) `x * y]`

   (e) ...

4. 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]
   ```

5. Mit welchen vordefinierten Funktionen in Haskell stimmen `f`, `g` und `h` aus der vorigen Aufgabe überein?

6. Welche der Funktionen `f`, `g` und `h` aus Aufgabe 4 sind rekursiv? Welche nicht? Welches Rekursionsmuster liegt bei den rekursiven Funktionen vor.

7. Was versteht man unter *Rechnen auf Parameterposition*? Illustrieren Sie Ihre Antwort mit einem Beispiel.

8. Beschreiben Sie die Liste `['a','e','i','o','u']` mithilfe einer Listenkomprehension.

9. Was besagt der *Hauptleitsatz fkt. Programmierung*?

10. Geben Sie je ein Beispiel an für eine:

    (a) schlicht

    (b) linear

    (c) baumartig

    (d) geschachtelt

    rekursive Funktion.

11. Was ist Rekursion auf

    (a) mikroskopischer

    (b) makroskopischer

    Ebene?

12. Sie haben für ein Problem ein exponentielles Lösungsverfahren. Ist es sinnvoller, einen neuen, schnelleren Rechner zu kaufen oder ein alternatives komplexitätsmßig günstigeres Lösungsverfahren zu finden? Begründen Sie Ihre Antwort.

13. Sie haben für ein Problem ein Lösungsverfahren einer bestimmten Komplexität. Ist es ein Königsweg, ein Lösungsverfahren geringerer Komplexität zu suchen, um schneller zu Ergebnissen zu gelangen? Begründen Sie Ihre Antwort.

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

```
uncurry :: (a -> b -> c)-> ((a,b) -> c)
uncurry g (x,y) = g x y
```

15. Was sind gültige Operatorabschnitte? Was nicht? Welche Bedeutung haben die gültigen Operatorabschnitte?

    (a) `(*10)`

    (b) `(10*)`

    (c) `((*) 10)`

    (d) `(10 (*))`

    (e) `(div 2)`

    (f) `(2 div)`

    (g) `(‘div‘ 2)`

    (h) `(2 ‘div‘)`

    (i) `(-1)`

    (j) `(1-)`

    (k) `((-)1)`

    (l) `(1(-))`

16. Warum liefern mit dem Schlüsselwort `type` eingeführte Namen keine höhere Typsicherheit?

17. Was versteht man unter einer Minimalvervollständigung einer Typklasse? Wofür ist eine Minimalvervollständigung gut?

18. Was ist automatische Typklasseninstanzbildung? Welche Voraussetzungen müssen vorliegen? Welche Grenzen oder Einschränkungen gibt es dafür?

19. `type` und `newtype`-Deklarationen leisten dasselbe, führen aber zu unterschiedlich performanten Implementierungen. Richtig oder falsch. Begründen Sie Ihre Antwort.

20. Aufzählungs- und Produkttypen sind Spezialfälle von Summentypen. In welcher Weise?

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

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

type Haustier = (Name,Lieblingsplatz,Hoertauf)
newtype Haustier = H (Name,Lieblingsplatz,HoertAuf)
```

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

    (a) Typsynonym?

    (b) Neuer Typ?

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

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

23. Listenkomprehensionen sttützen sich i.a. auf Generatoren, Transformatoren, Filter und Tests. Was davon wird in der Definition von `quickSort` genutzt?

```
quickSort :: Ord a => [a] -> [a]
quickSort []     = []
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                   ++ [x]
                   ++ quickSort [y | y <- xs, y > x]
```

24. Kann die Definition von `quickSort` vereinfacht werden zu?

```
quickSort :: Ord a => [a] -> [a]
quickSort []     = []
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                   ++ quickSort [y | y <- xs, y > x]
```

    Begründen Sie Ihre Antwort.

25. Skizzieren Sie die Evolution von Programmierparadigmen bzw. stilen. Wodurch ist sie angetrieben? Wodurch gekennzeichnet?