

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 14 vom 29.01.19: Rückblick & Ausblick

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ Monaden als Berechnungsmuster
 - ▶ Domänenspezifische Sprachen (DSLs)
 - ▶ Scala — Eine praktische Einführung
 - ▶ Rückblick & Ausblick

Organisatorisches

- ▶ Bitte für die Programmierübung (“E-Klausur”) anmelden (stud.ip)

- ▶ Bitte an der **Online-Evaluation** teilnehmen (stud.ip)

Inhalt

- ▶ E-Klausur

- ▶ Rückblick und Ausblick

Elektronische Programmierung

Erinnerung: Scheinkriterien

- ▶ Elektronische Klausur am Ende (Individualität der Leistung)
- ▶ Mind. 50% in allen Übungsblättern und mind. 50% in der E-Klausur
- ▶ Note = 50% Übungsblätter und 50% E-Klausur
- ▶ **Notenspiegel** (in Prozent aller Punkte):

Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	n/b

Aufbau

- ▶ Kleine **Programmierübungen**
 - ▶ Rahmen vorgegeben, mit kurzen Unit-Tests
 - ▶ Tests sind nicht vollständig — Erfüllung **notwendig** aber nicht **hinreichend**.
 - ▶ Ziel: Prüfung **elementarer Haskellkenntnisse** (Individualität der Prüfungsleistung)
- ▶ **Verständnisfragen**
 - ▶ Multiple-Choice-Tests
 - ▶ Zusatzaufgaben — Übung auch ohne Verständnisfragen zu bestehen
 - ▶ Ziel: Prüfung des **vertieften Verständnisses** des Stoffs
- ▶ Wertung: Programmierübung 80%, Verständnisfragen 20%

Beispiel Programmierübungen

Definieren Sie eine Funktion

```
ostern :: String → Int
```

die zählt, wie oft in einer Zeichenkette die Zeichenkette "ei" enthalten ist.

Beispiel:

```
ostern "ei , ei , oh , eiaiei" ~→ 4
```

Beispiel Programmierübungen

Definieren Sie eine Funktion `concatSnd`, welche eine Liste aus Paaren von Elementen und Listen auf eine Liste von Paaren von Elementen abbildet (also die Eingabelisten der zweiten Komponente konkateniert).

Beispiel:

```
concatSnd [(True, "xy"), (False, "foo")] ~>
  [(True, 'x'), (True, 'y'), (False, 'f'), (False, 'o'), (False, 'o')]
concatSnd [(1, [2, 3]), (7, [9, 5])] ~>
  [(1, 2), (1, 3), (7, 9), (7, 5)]
```

Beispiel Programmierübung

Eine Matrix ist als Liste ihrer Spaltenvektoren dargestellt:

```
data Matrix a = M [[a]]
```

Schreiben Sie eine Funktion

```
row :: Matrix a → Int → [a]
```

die die i -te Zeile (gezählt ab 1) einer Matrix zurückgibt.

Beispiel:

```
row (M [[3,7,5],[9,2,0],[5,8,1]]) 2 ~→ [7,2,8]
```

Beispiel Programmierübung

Definieren Sie eine Funktion

```
subseqs :: [a] → [[a]]
```

welche die nichtleeren Teillisten einer Liste berechnet.

Beispiel:

```
subseqs "pi3" ⇨ ["p", "pi", "pi3", "i", "i3", "3"]
```

Beispiel: Verständnisfrage

Betrachten Sie folgende Werte:

```
Otto  
Karl Otto "Heinz"  
Karl (Karl Otto [1,7]) "17"
```

Für welche der folgenden Typdeklarationen sind diese Werte wohlgetypt:

- data** T a = Otto | Karl (T a) [a]
- data** T a b = Otto | Karl a b
- data** T a = Otto | Karl (T a) String
- data** T a b = Otto a | Karl b [a]

Beispiel: Verständnisfrage

Betrachten Sie folgende Werte:

```
Otto  
Karl Otto "Heinz"  
Karl (Karl Otto [1,7]) "17"
```

Für welche der folgenden Typdeklarationen sind diese Werte wohlgetypt:

- data** T a = Otto | Karl (T a) [a]
- data** T a b = Otto | Karl a b
- data** T a = Otto | Karl (T a) String
- data** T a b = Otto a | Karl b [a]

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
count :: Ord a => a -> Tree a -> Int
```

```
count _ Leaf = 0
```

```
count a (Node l b r) | a < b = count a l  
                    | a == b = 1 + count a r  
                    | a > b = count a r
```

Welche der folgenden Invarianten eines Baumes `Node l a r` stellt sicher, dass `count` korrekt ist:

- `a` ist kleiner-gleich allem in `l` und größer-gleich allem in `r`
- `a` ist größer-gleich allem in `l` und kleiner-gleich allem in `r`
- `a` ist größer als alles in `l` und kleiner-gleich allem in `r`
- `a` ist kleiner-gleich allem in `l` und größer als alles in `r`

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
count :: Ord a => a -> Tree a -> Int
```

```
count _ Leaf = 0
```

```
count a (Node l b r) | a < b = count a l  
                    | a == b = 1 + count a r  
                    | a > b = count a r
```

Welche der folgenden Invarianten eines Baumes `Node l a r` stellt sicher, dass `count` korrekt ist:

- `a` ist kleiner-gleich allem in `l` und größer-gleich allem in `r`
- `a` ist größer-gleich allem in `l` und kleiner-gleich allem in `r`
- `a` ist größer als alles in `l` und kleiner-gleich allem in `r`
- `a` ist kleiner-gleich allem in `l` und größer als alles in `r`

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
count :: Ord a => a -> Tree a -> Int
```

```
count _ Leaf = 0
```

```
count a (Node l b r) | a < b = count a l  
                    | a == b = 1 + count a r  
                    | a > b = count a r
```

Welche der folgenden Eigenschaften erfüllt count?

- count ist **injektiv**
- count ist **total**
- count ist **partiell**
- count ist **strikt** im **ersten** Argument
- count ist **strikt** im **zweiten** Argument

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
count :: Ord a => a -> Tree a -> Int
```

```
count _ Leaf = 0
```

```
count a (Node l b r) | a < b = count a l  
                    | a == b = 1 + count a r  
                    | a > b = count a r
```

Welche der folgenden Eigenschaften erfüllt count?

- count ist **injektiv**
- count ist **total**
- count ist **partiell**
- count ist **strikt** im **ersten** Argument
- count ist **strikt** im **zweiten** Argument

Beispiel: Verständnisfrage

Gegeben folgende Funktionsdefinition:

```
f :: a -> [a] -> [a]
f a (b:bs) = b: f a bs
f a []     = [a]
```

Welche Definitionen sind **äquivalent**?

- f1 x xs = foldr (:) xs [x]
- f2 = (flip (++) o (: []))
- f3 x xs = foldl (flip (:)) xs x
- f4 a = (++ [a])

Beispiel: Verständnisfrage

Gegeben folgende Funktionsdefinition:

```
f :: a -> [a] -> [a]
f a (b:bs) = b: f a bs
f a []     = [a]
```

Welche Definitionen sind **äquivalent**?

- f1 x xs = foldr (:) xs [x]
- f2 = (flip (++) o (: []))
- f3 x xs = foldl (flip (:)) xs x
- f4 a = (++) [a]

Rückblick und Ausblick

Warum funktionale Programmierung lernen?

- ▶ Funktionale Programmierung macht aus Programmierern Informatiker
- ▶ Blick über den Tellerrand — was kommt in 10 Jahren?
- ▶ **Herausforderungen** der Zukunft
- ▶ Enthält die **wesentlichen** Elemente moderner Programmierung

Zusammenfassung Haskell

Stärken:

- ▶ Abstraktion durch
 - ▶ Polymorphie und Typsystem
 - ▶ algebraische Datentypen
 - ▶ Funktionen höherer Ordnung
- ▶ Flexible Syntax
- ▶ Haskell als Meta-Sprache
- ▶ Ausgereifter Compiler
- ▶ Viele Büchereien

Schwächen:

- ▶ Komplexität
- ▶ Büchereien
 - ▶ Nicht immer gut gepflegt
- ▶ Viel im Fluß
 - ▶ Kein stabiler und brauchbarer Standard
- ▶ Divergierende Ziele:
 - ▶ Forschungsplattform **und** nutzbares Werkzeug

Andere Funktionale Sprachen

- ▶ **Standard ML (SML):**
 - ▶ Streng typisiert, strikte Auswertung
 - ▶ Standardisiert, formal definierte Semantik
 - ▶ Drei aktiv unterstützte Compiler
 - ▶ Verwendet in Theorembeweisern (Isabelle, HOL)
 - ▶ <http://www.standardml.org/>
- ▶ **Caml, O'Caml:**
 - ▶ Streng typisiert, strikte Auswertung
 - ▶ Hocheffizienter Compiler, byte code & nativ
 - ▶ Nur ein Compiler (O'Caml)
 - ▶ <http://caml.inria.fr/>

Andere Funktionale Sprachen

▶ **LISP** und **Scheme**

- ▶ Ungetypt/schwach getypt
- ▶ Seiteneffekte
- ▶ Viele effiziente Compiler, aber viele Dialekte
- ▶ Auch industriell verwendet

▶ **Hybridsprachen:**

- ▶ Scala (Functional-OO, JVM)
- ▶ F# (Functional-OO, .Net)
- ▶ Clojure (Lisp, JVM)

Was spricht gegen funktionale Programmierung?

- ▶ Mangelnde Unterstützung:
 - ▶ Libraries, Dokumentation, Entwicklungsumgebungen
 - ▶ Wird besser (Scala)...
- ▶ Programmierung nur kleiner Teil der SW-Entwicklung
- ▶ Nicht verbreitet — funktionale Programmierer zu teuer
- ▶ Konservatives Management
 - ▶ “Nobody ever got fired for buying IBM”

Was spricht gegen funktionale Programmierung?

- ▶ Mangelnde Unterstützung:
 - ▶ Libraries, Dokumentation, Entwicklungsumgebungen
 - ▶ Wird besser (Scala)...
- ▶ Programmierung nur kleiner Teil der SW-Entwicklung
- ▶ Nicht verbreitet — funktionale Programmierer zu teuer
- ▶ Konservatives Management
 - ▶ “Nobody ever got fired for buying SAP”

Haskell in der Industrie

- ▶ Simon Marlow bei Facebook: Sigma — Fighting spam with Haskell
- ▶ Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- ▶ Bluespec: Schaltkreisentwicklung, DSL auf Haskell-Basis
- ▶ Galois, Inc: Cryptography (Cryptol DSL)
- ▶ Siehe auch: Haskell in Industry

Funktionale Programmierung in der Industrie

- ▶ **Scala:**
 - ▶ Twitter, Foursquare, Guardian, . . .
- ▶ **Erlang**
 - ▶ schwach typisiert, nebenläufig, strikt
 - ▶ Fa. Ericsson (Telekom-Anwendungen), WhatsApp
- ▶ **FL**
 - ▶ ML-artige Sprache
 - ▶ Chip-Verifikation der Fa. Intel
- ▶ **Python** (und andere Skriptsprachen):
 - ▶ Listen, Funktionen höherer Ordnung (map, fold), anonyme Funktionen, Listenkomprehension

Perspektiven funktionaler Programmierung

▶ **Forschung:**

- ▶ Ausdrucksstärkere Typsysteme
- ▶ für effiziente Implementierungen
- ▶ und eingebaute Korrektheit (Typ als Spezifikation)
- ▶ Parallelität?

▶ **Anwendungen:**

- ▶ Eingebettete domänenspezifische Sprachen
- ▶ Zustandsfreie Berechnungen (MapReduce, Hadoop, Spark)
- ▶ **Big Data** and **Cloud Computing**

If you liked this course, you might also like ...

- ▶ Die Veranstaltung **Reaktive Programmierung** (Sommersemester 2019)
 - ▶ Scala, nebenläufige Programmierung, fortgeschrittene Techniken der funktionalen Programmierung
- ▶ Wir suchen **studentische Hilfskräfte** am DFKI, FB CPS
 - ▶ Scala als Entwicklungssprache
- ▶ Wir suchen **Tutoren für PI3**
 - ▶ Im WS 2019/20 — **meldet Euch** bei Thomas Barkowsky (oder bei mir)!

Tschüß!

