Praktische Informatik 3: Funktionale Programmierung

Vorlesung 14 (31.01.23): Rückblick und Ausblick

Christoph Lüth

Wintersemester 2022/23

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
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick & Ausblick

Organisatorisches

- Bitte für die Programmierübung (“E-Klausur”) anmelden (stud.ip)
- Bitte an der **Online-Evaluation** teilnehmen (stud.ip)

I. Elektronische Programmierübung

Erinnerung: Scheinkriterien

- Mindestens 50% in den Einzelübungsblättern,
  in allen Übungsblättern und
  mindestens 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**

Elektronische Klausur

- **Termin**: 13.03.2023, 14:00– 15:30 und 15:45– 17:15
- **Ort**: Testzentrum am Boulevard neben der Bibliothek
- **Dauer**: 90 Minuten
- **Ablauf**:
  - Einfache Programmierübungen in der Art der Übungsaufgaben
  - Einige Multiple-Choice Fragen als **Bonus**

Aufbau

- Kleine **Programmierübungen**
  - Rahmen vorgegeben, mit kurzen Unit-Tests
  - Tests sind nicht vollständing — 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: Klausur – 20 Punkte, Verständnisfragen – 5 Punkte

Beispiel Programmierübungen

Definieren Sie eine 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 :: [(a, [b])] → [(a, b)]
```

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 wir folgende Funktionsdefinition:

```
fun x y z = y z
```

Welche der folgenden Typsignaturen wären für diese Definition typkorrekt?

- ☐ `fun :: a→ (c→ b)→ c→ b`
- ☐ `fun :: Int→ ([a]→ Int)→ [a]→ Int`
- ☐ `fun :: a→ b→ c→ b`
- ☐ `fun :: Int→ (b→ c)→ Int→ b`

Beispiel: Verständnisfrage

Betrachten wir folgende Funktionsdefinition:

```
fun x y z = y z
```

Welche der folgenden Typsignaturen wären für diese Definition typkorrekt?

- ☑ `fun :: a→ (c→ b)→ c→ b`
- ☑ `fun :: Int→ ([a]→ Int)→ [a]→ Int`
- ☐ `fun :: a→ b→ c→ b`
- ☐ `fun :: Int→ (b→ c)→ Int→ b`

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 folgenden fehlerhaften Definitionsversuch eines algebraischen Datentypen:

```
data Foo a b = Foo [a] (Foo c Int)
             | bar (Foo Int a)
```

Welche der folgende Aussagen beschreibt tatsächliche Fehler in dieser Definition:

- ☐ Die Typvariable `c` ist nicht definiert.
- ☐ Der Konstruktur `bar` ist kleingeschrieben.
- ☐ Der Konstruktor `Foo` heißt genauso wie der Datentyp.
- ☐ Die Typvariable `b` wird auf der rechten Seite der Definition nicht genutzt.

Beispiel: Verständnisfrage

Betrachten Sie folgenden fehlerhaften Definitionsversuch eines algebraischen Datentypen:

```
data Foo a b = Foo [a] (Foo c Int)
             | bar (Foo Int a)
```

Welche der folgende Aussagen beschreibt tatsächliche Fehler in dieser Definition:

- ☑ Die Typvariable `c` ist nicht definiert.
- ☑ Der Konstruktur `bar` ist kleingeschrieben.
- ☐ Der Konstruktor `Foo` heißt genauso wie der Datentyp.
- ☐ Die Typvariable `b` wird auf der rechten Seite der Definition nicht genutzt.

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 (++) ∘ (:[]))`
- ☐ `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 (++) ∘ (:[]))`
- ☐ `f3 x xs = foldl (flip (:)) xs x`
- ☑ `f4 a = (++ [a])`

Vorbereitung und Durchführung

- Auf der Webseite sind alte Klausuren verfügbar.
- Für ein **realistisches** Übungsszenario:
  - 90 Minuten Zeit für die Klausuren
  - Windows-10 Rechner, Visual Studio Code
  - Kein Internet (vorher einmal `stack` starten)

II. 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 und integriert
- Nur ein ernsthafter **Compiler**
- Divergierende Ziele:
  - Forschungsplattform **und** nutzbares Werkzeug

Andere Funktionale Sprachen

- **Standard ML** (SML):
  - Streng typisiert, strikte Auswertung
  - Standardisiert, formal definierte Semantik
  - Mehrere aktiv (?) unterstütze 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)
  - Elixir (Erlang VM)

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 Meta (Facebook), Simon Peyton-Jones bei Microsoft.
- secuCloud in Hamburg (`https://www.secucloud.com/`), now part of Aryaka
- Schaltkreisentwicklung:
  - Bluespec, DSL auf Haskell-Basis; Clash, Haskell mit abhängigen Typen
  - Chisel und SpinalHDL: in Scala eingebettet DSLs
- Galois, Inc: Cryptography (Cryptol DSL)
- Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- Siehe auch: Haskell in Industry (`https://wiki.haskell.org/Haskell_in_industry`)
- Andere Sprachen: Scala, Erlang, Elm, ...

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** (findet irregulär statt)
  - Scala, nebenläufige Programmierung, fortgeschrittene Techniken der funktionalen Programmierung
- Wir suchen **studentische Hilfskräfte** am DFKI, FB CPS
  - Scala und Haskell als Entwicklungssprachen
- Wir suchen **Tutoren für PI3**
  - Im WS 2023/24 — **meldet Euch** bei Thomas Barkowsky (oder bei mir)!

Tschüß!