Praktische Informatik 3: Funktionale Programmierung
Vorlesung 14 vom 29.01.13: Schlußbemerkungen

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Effizienzaspekte
  - Eine Einführung in Scala
  - Rückblich & Ausblick

Fachgespräch: Beispielaufgabe

Fachgespräch: Beispielaufgabe

Definieren Sie eine Funktion leastSpaces, welche aus einer Liste von Zeichenketten diejenige zurückgibt, welche die wenigsten Leerzeichen enthält.

Beispiel:

leastSpace ["a␣bc", "pqr", "x␣y␣z"] $\rightsquigarrow$ "pqr"

Inhalt

- Wiederholung
- Rückblick, Ausblick

Verständnisfragen

Auf allen Übungsblättern finden sich Verständnisfragen zur Vorlesung. Diese sind nicht Bestandteil der Abgabe, können aber im Fachgesprach thematisiert werden. Wenn Sie das Gefühl haben, diese Fragen nicht sicher beantworten zu können, wenden Sie sich gerne an Ihren Tutor, an Berthold Hoffmann in seiner Fragestunde, oder an den Dozenten.

Verständnisfragen: Übungsblatt 1

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

Verständnisfragen: Übungsblatt 1

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

2. Was ist ein algebraischer Datentyp, und was ist ein Konstruktor?

Verständnisfragen: Übungsblatt 1

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

2. Was ist ein algebraischer Datentyp, und was ist ein Konstruktor?

3. Was sind die drei Eigenschaften, welche die Konstruktoren eines algebraischen Datentyps auszeichnen, was ermöglichen sie und warum?

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

2. Was ist der Unterschied zwischen Bäumen und Graphen, in Haskell modelliert?

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

2. Was ist der Unterschied zwischen Bäumen und Graphen, in Haskell modelliert?

3. Was sind die wesentlichen Gemeinsamkeiten, und was sind die wesentlichen Unterschiede zwischen algebraischen Datentypen in Haskell, und Objekten in Java?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

2. Welche zwei Arten der Polymorphie haben wir kennengelernt, und wie unterschieden sie sich?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

2. Welche zwei Arten der Polymorphie haben wir kennengelernt, und wie unterschieden sie sich?

3. Was ist der Unterschied zwischen Polymorphie in Haskell, und Polymorphie in Java?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

3. Was ist η-Kontraktion, und warum ist es zulässig?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

3. Was ist η-Kontraktion, und warum ist es zulässig?

4. Wann verwendet man foldr, wann foldl, und unter welchen Bedingungen ist das Ergebnis das gleiche?

Verständnisfragen: Übungsblatt 5

1. foldr ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

Verständnisfragen: Übungsblatt 5

1. foldr ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

2. Wann kann foldr f a xs auch für ein zyklisches Argument xs (bspw. eine zyklische Liste) terminieren?

Verständnisfragen: Übungsblatt 5

1. foldr ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

2. Wann kann foldr f a xs auch für ein zyklisches Argument xs (bspw. eine zyklische Liste) terminieren?

3. Was ist die Grundidee hinter Parserkombinatoren, und funktioniert diese Idee nur für Parser oder auch für andere Problemstellungen?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

2. Welches sind drei charakteristische Eigenschaften von Haskells Typsystem (Hindley-Milner)?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

2. Welches sind drei charakteristische Eigenschaften von Haskells Typsystem (Hindley-Milner)?

3. Was ist Unifikation, und welche Rolle spielt sie bei der Typinferenz? Wann kann sie fehlschlagen, und was sind Beispiele (Haskell-Programme, deren Typinferenz fehlschlägt) dafür?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

2. Was sind Unterschiede und Gemeinsamkeiten zwischen ADTs und Objekten, wie wir sie aus Sprachen wie Java kennen?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

2. Was sind Unterschiede und Gemeinsamkeiten zwischen ADTs und Objekten, wie wir sie aus Sprachen wie Java kennen?

3. Wozu dienen Module in Haskell?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

2. Warum „finden Tests Fehler“, aber „zeigen Beweise Korrektheit“, wie in der Vorlesung behauptet? Stimmt das immer?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

2. Warum „finden Tests Fehler“, aber „zeigen Beweise Korrektheit“, wie in der Vorlesung behauptet? Stimmt das immer?

3. Müssen Axiome immer ausführbar sein? Welche Axiome wären nicht ausführbar?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als

```
data Stream α = Cons α (Stream α)
```

   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als

   ```
   data Stream α = Cons α (Stream α)
   ```

   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als

   ```
   data Stream α = Cons α (Stream α)
   ```

   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

3. Wie kann man in einem Induktionsbeweis die Induktionsvoraussetzung stärken?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als

   ```
   data Stream α = Cons α (Stream α)
   ```

   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

3. Wie kann man in einem Induktionsbeweis die Induktionsvoraussetzung stärken?

4. Gibt es einen Weihnachtsmann?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

2. Warum sind Aktionen nicht explizit als Zustandsübergang modelliert, sonder als abstrakter Datentyp IO?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

2. Warum sind Aktionen nicht explizit als Zustandsübergang modelliert, sonder als abstrakter Datentyp IO?

3. Außer dem Mangel an referentieller Transparenz, was ist die entscheidende Eigenschaft, die Aktionen von reinen Funktionen unterscheidet?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

2. Ist eine in allen Argumenten als strikt erkannte und übersetzte Funktion immer schneller in der Ausführung als dieselbe als nicht-strikt übersetzte Funktion?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

2. Ist eine in allen Argumenten als strikt erkannte und übersetzte Funktion immer schneller in der Ausführung als dieselbe als nicht-strikt übersetzte Funktion?

3. Warum kann ich die Funktion seq nicht in Haskell definieren?

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
- Viel im Fluß
  - Kein stabiler und brauchbarer Standard
- Divergierende Ziele:
  - Forschungsplattform und nutzbares Werkzeug
- Entwicklungsumgebungen

Andere Funktionale Sprachen

- Standard ML (SML):
  - Streng typisiert, strikte Auswertung
  - Formal definierte Semantik
  - Drei 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)

Warum funktionale Programmierung nie Erfolg haben wird

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

Funktionale Programmierung in der Industrie

- Erlang
  - schwach typisiert, nebenläufig, strikt
  - Fa. Ericsson — Telekom-Anwendungen
- `FL`
  - ML-artige Sprache
  - Chip-Verifikation der Fa. Intel
- Python (und andere Skriptsprachen):
  - Listen, Funktionen höherer Ordnung (map, fold), anonyme Funktionen, Listenkomprehension
- Big Data:
  - Zustandsfreie Berechnungen (MapReduce, Hadoop)

Warum funktionale Programmierung lernen?

- Denken in Algorithmen, nicht in Programmiersprachen
- Abstraktion: Konzentration auf das Wesentliche
- Wesentliche Elemente moderner Programmierung:
  - Datenabstraktion und Funktionale Abstraktion
  - Modularisierung
  - Typisierung und Spezifikation
- Blick über den Tellerrand — Blick in die Zukunft
- Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?

Hilfe!

- Haskell: primäre Entwicklungssprache am DFKI, FG CPS
  - Sicherheit in der Robotik: `http://www.dfki.de/cps/sams`
  - Formale Programmentwicklung: `http://www.tzi.de/cofi/hets`
- Wir suchen studentische Hilfskräfte für diese Projekte
- Wir bieten:
  - Angenehmes Arbeitsumfeld
  - Interessante Tätigkeit
- Wir suchen Tutoren für PI3
  - im WS 13/14 — meldet Euch bei Berthold Hoffmann!

Tschüß!