Tutorium 02: Kombinatoren, Lazyness

David Kaufmann

09. November 2022

Tutorium Programmierparadigmen am KIT

Verwirrung vom letzten Mal

Wir waren uns nicht sicher was wie stark bindet

```
module Length where
lengthL l = lengthAcc l 0 where
  lengthAcc [] a = a
  lengthAcc (h:t) a = lengthAcc t a + 1
```

```
lengthAcc [1,2] 0
```

```
=> (lengthAcc [2] 0) + 1
```

```
=> ((lengthAcc [] 0) + 1) + 1
```

Notenskala

- -
- Richtig, kleine Fehler
- Aufgabe nicht verstanden
- Grundansatz falsch
- Richtig!
- Richtiger Ansatz, aber unvollständig

Übungsblatt 1

Stabile Sortieralgorithmen

Wir sortieren diese Liste anhand dem ersten Element der Tupel

- `[(1,1),(4,2),(2,1),(5,1),(4,1)]`

Das ist stabil

- `[(1,1),(2,1),(4,2),(4,1),(5,1)]`

Das nicht

- `[(1,1),(2,1),(4,1),(4,2),(5,1)]`

$\rightsquigarrow$ Elemente mit dem gleichen Key müssen ihre Reihenfolge beibehalten

Vorlesungswiederholung

Listenkombinatoren

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  - Bsp.: `foldr f s [1,2,3]` berechnet `(f 1 (f 2 (f 3 s)))`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
  - Bsp.: `foldl f s [1,2,3]` berechnet `(f (f (f s 1) 2) 3)`
- Für beide gilt: `foldr operation initial list`

Listenkombinatoren

Weitere Kombinatoren:

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `zip :: [a] -> [b] -> [(a, b)]`
- `and, or :: [Bool] -> Bool`

Idee: Statt Rekursion selbst zu formulieren verwenden wir fertige „Bausteine“, sogenannte „Kombinatoren“.

Aufgaben

Schreibt ein Modul `Tut02` mit:

- `import Prelude hiding (foldl, foldr, map, filter, scanl, zip, zipWith)`
- `map` — Einmal von Hand, einmal per Fold
- `filter` — Einmal von Hand, einmal per Fold
- `squares l` — Liste der Quadrate der Elemente von `l`
- `zip as bs` — Erstellt Tupel der Elemente von `as` und `bs`
- `zipWith f as bs` — Wendet komponentenweise `f` auf die Elemente von `as` und `bs` an
  - Bspw. `zipWith (+) [1, 1, 2, 3] [1, 2, 3, 5] == [2, 3, 5, 8]`

Aufgaben

Schreibt ein Modul `Tut02` mit:

- `import Prelude hiding (foldl, foldr, map, filter, scanl, zip, zipWith)`
- `map` — Einmal von Hand, einmal per Fold
- `filter` — Einmal von Hand, einmal per Fold
- `squares l` — Liste der Quadrate der Elemente von `l`
- `zip as bs` — Erstellt Tupel der Elemente von `as` und `bs`
- `zipWith f as bs` — Wendet komponentenweise `f` auf die Elemente von `as` und `bs` an
  - Bspw. `zipWith (+) [1, 1, 2, 3] [1, 2, 3, 5] == [2, 3, 5, 8]`
- `foldl`
- `scanl f i l` — Wie `foldl`, gibt aber eine Liste aller Akkumulatorwerte zurück
  - Bspw. `scanl (*) 1 [1, 3, 5] == [1, 3, 15]`

Lazy Evaluation

```
$ ghci
GHCi, version 8.8.4: http://www.haskell.org/ghc/
Prelude> x = 42 `div` 0
Prelude> putStrLn $ show x
*** Exception: divide by zero
```

- Was heißt Lazy Evaluation?
- Wieso tritt erst bei der zweiten Eingabe ein Fehler auf?

Lazy Evaluation

```
$ ghci
GHCi, version 8.8.4: http://www.haskell.org/ghc/
Prelude> x = 42 `div` 0
Prelude> putStrLn $ show x
*** Exception: divide by zero
```

- Was heißt Lazy Evaluation?
- Wieso tritt erst bei der zweiten Eingabe ein Fehler auf?
- $\rightsquigarrow$ Berechnungen finden erst statt, wenn es *absolut* nötig ist

Lazy Evaluation

wiki.haskell.org/Lazy_Evaluation:

> *Lazy evaluation means that expressions are not evaluated when they are bound to variables, but their evaluation is* **deferred** *until their results are needed by other computations.*

- Auch: *call-by-name* im Gegensatz zu *call-by-value* in bspw. C
- Was bringt das?

Lazy Evaluation

wiki.haskell.org/Lazy_Evaluation:

> *Lazy evaluation means that expressions are not evaluated when they are bound to variables, but their evaluation is* **deferred** *until their results are needed by other computations.*

- Auch: *call-by-name* im Gegensatz zu *call-by-value* in bspw. C
- Was bringt das?
- Ermöglicht bspw. arbeiten mit unendlichen Listen
- Berechnungen, die nicht gebraucht werden, werden nicht ausgeführt

Lazy Evaluation

- Arithmetische Operatoren `((+), (-), (*))` werten Argumente **immer** aus
- Vergleichsoperatoren `(<), (<=), (==)` werten Argumente **immer** aus
- Boolsche Operatoren `((), (||))` nutzen Short-Circuit-Auswertung
- Strukturelle Operatoren `((:), (,))` werten Argumente **nie** aus

List Comprehension

Sytnax: `[e |` $q_1, ..., q_m$`]`

- `e` ist ein Ausdruck
- $q_1, ..., q_m$ sind Generatoren (`p <- xs`) oder Prädikate

List Comprehension

Sytnax: `[e|`$q_1, ..., q_m$`]`

- `e` ist ein Ausdruck
- $q_1, ..., q_m$ sind Generatoren (`p <- xs`) oder Prädikate

Die Reihenfolge ist wichtig!

- `[(x,y)| x <- [1..10], y <- [1..x]]` Funktioniert
- `[(x,y)| y <- [1..x], x <- [1..10]]` Kompilierfehler
- `[(x,y)| x <- [1..], y <- [1..]]` Immer `(1,n)`

Aufgabe

Erstelle eine Liste, die alle Tupel `(x,y)` mit $x, y \in \mathbb{N}_+$ genau einmal enthält.

Aufgabe

Erstelle eine Liste, die alle Tupel `(x,y)` mit $x, y \in \mathbb{N}_+$ genau einmal enthält.

```
[(x,y)|n <- [1..], x <- [1..n-1], y <- [n-x]]
```

Cheatsheet: Tupel und Konzepte

- *List comprehension*, *Laziness*
- `[f x | x <- xs, p x]` ≡ `map f (filter p xs)`
  Bspw.: `[x * x | x <- [1..]]` ⇒ `[1,4,9,16,25,...]`
- *Tupel*
- `(,) :: a -> b -> (a, b)` („Tupel-Konstruktor“)
- `fst :: (a, b) -> a`
- `snd :: (a, b) -> b`