Tutorium 02: Mehr Haskell

Paul Brinkmeier

17. November 2020

Tutorium Programmierparadigmen am KIT

Notenskala

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

Heutiges Programm

Programm

- Übungsblatt 1/Aufgabe 1
- Wiederholung der Vorlesung
- Aufgaben zu Haskell

Übungsblatt 1

1.1, 1.2 — `pow1` und `pow2`

```
module Arithmetik1 where

pow1 b e | e < 0      = error "e negativ"
         | e == 0     = 1
         | otherwise = b * pow1 b (e-1)

pow2 b e
  | e < 0           = error "e negativ"
  | e == 0          = 1
  | e `mod` 2 == 0 =     pow2 (b * b) (e `div` 2)
  | otherwise       = b * pow2 (b * b) (e `div` 2)
```

1.3 — pow3

```
module Arithmetik2 where


pow3 b e
  | e < 0 = error "e negativ"
  | otherwise = pow3Acc 1 b e
  where
    pow3Acc acc b e
      | e == 0 = acc
      | e `mod` 2 == 0 =
        pow3Acc acc (b * b) (e `div` 2)
      | e `mod` 2 == 1 =
        pow3Acc (b * acc) (b * b) (e `div` 2)
```

1.4 — root

```
module Arithmetik3 where

import Arithmetik2 (pow3)

root exp r
  | exp <= 0  = error "Exponent negativ"
  | r < 0     = error "Wurzel komplex"
  | otherwise = searchRoot 0 (r + 1)
  where
    searchRoot lower upper
      | upper - lower == 1 = lower
      | r < avg `pow3` exp = searchRoot lower avg
      | otherwise          = searchRoot avg upper
      where avg = (lower + upper) `div` 2
```

1.5 — isPrime

```
module Arithmetik4 where

import Arithmetik3 (root)

isPrime n = not (any (divides n) [2..root 2 n])
  where
    divides p q = p `mod` q == 0
```

Aufgaben 2 und 3

Teilaufgaben 2 und 3 sind noch nicht korrigiert.

$\rightsquigarrow$ nächste Woche

Wiederholung: Funktionen

Cheatsheet: Listenkombinatoren

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `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“.

Cheatsheet: Tupel und Konzepte

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

Cheatsheet: Typen

- `Char`, `Int`, `Integer`, ...
- `String`
- Typvariablen/Polymorphe Typen:
  - `(a, b)`: Tupel
  - `[a]`: Listen
  - `a -> b`: Funktionen
  - Vgl. Java: `List<A>`, `Function<A, B>`
- Typsynonyme: `type String = [Char]`

Aufgaben

Schreibt ein Modul `Tut02` mit:

- `import Prelude (foldl, foldr)`
- `map` — Einmal von Hand, einmal per Fold
- `filter` — Einmal von Hand, einmal per Fold
- `squares l` — Liste der Quadrate der Elemente von `l`
- `odd, even` — Prüft ob eine Zahl (un-)gerade ist
- `odds, evens` — Liste aller (un-)geraden Zahlen `>= 0`
- `foldl`
- `scanl f l` — Wie `foldl`, gibt aber eine Liste aller Akkumulatorwerte zurück
  - Bspw. `scanl (*) 1 [1, 3, 5] == [1, 3, 15]`

Lazy Evaluation

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

Digits

Digits

Schreibt ein Modul `Digits` mit:

- `digits :: Int -> [Int]` — Liste der Stellen einer positiven Zahl.

Bspw.:

```
digits   42 == [4, 2]
digits  101 == [1, 0, 1]
digits 1024 == [1, 0, 2, 4]
digits    0 == [0]
digits (-5) == error "need positive number"
```

Hangman

Hangman

- pbrinkmeier.de/Hangman.hs
- `showHangman` — Zeigt aktuellen Spielstand als `String`
- `updateHangman` — Bildet Usereingabe (als `String`) und alten Zustand auf neuen Zustand ab
- `initHangman` — Anfangszustand, leere Liste

Hangman — CLI-Framework

```
module CLI where


runConsoleGame ::
  (s -> String) ->
  (String -> s -> s) ->
  s ->
  IO ()
```

- `s` ist der Typ des Spielzustands
- Anfänglicher Zustand: `[]` — leere Liste an Rateversuchen
- `showHangman :: [Char] -> String`
- `updateHangman :: String -> [Char] -> [Char]`
- `initHangman :: [Char]`

Hangman — Beispiele

- `showHangman "test" "e"` $\Rightarrow$ `". e . . | e"`
- `showHangman "test" "sf"` $\Rightarrow$ `". . s . | s f"`
- `updateHangman "f" "abc"` $\Rightarrow$ `"fabc"`