Haskell - Eine Einführung

Matthias Beyer

Furtwangen University

matthias.beyer@hs-furtwangen.de
mail@beyermatthias.de

8. Mai 2014

Overview

Was, wie, wo, wann, wer?

- Compilersprache
- Rein funktional
- 1990 erschienen
- Statisch typisiert
- nicht-strikt

Wikipedia

Beeinflusst von

- APL
- LISP
- Miranda
- ML

Wikipedia

Beeinflusst

- Agda
- Cayenne
- Clean
- Curry
- Python
- Scala
- C#
- F#

Wikipedia

Funktionale Sprachen

- Haskell
- Scala
- Erlang
- Scheme
- OCaml
- F-Sharp
- Makefile

Eigenschaften

- Funktionen geben Werte zurück, ändern aber keine Zustände
- Keine Imperativen Sprachkonstrukte

Wikipedia

Eigenschaften

- Keine Operationen, die Variablen verändern
- Beweise sind einfach, da Nebeneffekte fehlen
- Lazyness = nicht-strikt

Wikipedia

Einfache Funktionen

```
1 first x y = x
```

$$f(x, y) = x \tag{1}$$

Einfache Funktionen

```
1 quadrat x = x * x
```

$$f(x) = x * x = x^2 \tag{2}$$

Typen von Funktionen

```
1 quadrat :: Int -> Int
  quadrat :: a -> b
```

```
toUpper :: (Integral a) => a -> b
```

```
1 toUpper :: [Char] -> [Char]
```

Höhere Funktionen

- Funktionen bekommen Funktionen als Parameter

Problem: Array mit Ints, alle im quadrat.

$$f(a, g(x)) = \forall z \in a : g(z) \tag{3}$$

Lösung in C mit Funktion höherer Ord.

```c
int* map(int* ary, int c, int (*f)(int)) {
    for(int i = 0; i < c; i++)
        ary[i] = f(ary[i]);
}
```

```c
ary = map(ary, arylen, quadrat);
```

Lösung in Haskell

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Note: Rekursiv!

```haskell
map quadrat [1,2,3]
```

Pattern Matching

Pattern matching

Pattern Matching

```
1 lucky :: (Integral a) => a -> String
  lucky 7 = ``Lucky number seven''
3 lucky x = ``Sorry, nope...''
```

Pattern Matching

```haskell
fact :: (Integral a) => a -> a
fact 0 = 1
fact n = n * fact (n - 1)
```

$$f(0) = 1 \tag{4}$$

$$f(x) = x * f(x - 1) \tag{5}$$

Pattern Matching

```
charName :: Char -> String
charName 'a' = ''Albert``
charName 'b' = ''Berta``
charName 'c' = ''Christina``
charName 'd' = ''Daniel``
```

Tupel

```haskell
(1, 2, 3) -- A tupel of three

firstofthree :: (a, b, c) -> a
firstofthree (x, _, _) = x
```

$$f(t) = t_1, |t| = 3 \tag{6}$$

Guards

Guards

Guards

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = ``You're underweight''
    | bmi <= 25.0 = ``You're normal''
    | bmi <= 30.0 = ``You're fat''
    | otherwise = ``You crashed the system''
```

Guards

```
bmiTell :: (RealFloat a) => a -> String
bmiTell weight height
    | (weight / height ^ 2) <= 18.5 = ``You're underwe
    | (weight / height ^ 2) <= 25.0 = ``You're normal''
    | (weight / height ^ 2) <= 30.0 = ``You're fat''
    | otherwise = ``You crashed the system''
```

Typen

```haskell
data Bool = True | False
```

Also auch ungefähr:

```haskell
data Int = -2147483648 |...| -1 | 0 | 1 |...| 2147483
```

Typen

```haskell
data Shape = Circle Float Float Float |
     Rectangle Float Float Float Float
```

Surface berechnen:

```haskell
sf :: Shape -> Float

sf (Circle _ _ r) = pi * r ^ 2

sf (Rectangle x1 y1 x2 y2) =
    (abs $ x2-x1)*(abs $ y2-y1)
```

Typen

Surface berechnen:

```
sf $ Circle 10 20 10 -- 314.15927
sf $ Rectangle 0 0 100 100 -- 10000.0
```

Typen

Typen mappen:

```
map (Circle 10 20) [4, 5] -- 2 params, 3. gemappt
-- [Circle 10.0 20.0 4.0, Circle 10.0 20.0 5.0]
```

Typen

Wir verbessern unsern Typ:

```
data Point = Point Float Float
data Shape = Circle Point Float |
     Rectangle Point Point
```

Typen

Wir verbessern unsere Funktion:

```
1 sf :: Shape -> Float
  sf (Circle _ r) = pi * r ^ 2
3 sf (Rectangle (Point x2 y2) (Point x2 y2)) =
    (abs $ x2 - x1) * (abs $ y2 - y1)
```

Typen

Und nutzen sie:

```
sf (Rectangle (Point 0 0) (Point 100 100)) -- 10000.0
2
sf (Circle (Point 0 0) 24) -- 1809.5574
```

Typen

Typdefinition in schön:

```
1 data Person = Person { firstName :: String
                       , lastName :: String
3                      , age :: Int
                       }
```

Typen

Haskell hilft:

```
firstName :: Person -> String -- Schon definiert!
lastName :: Person -> String -- Schon definiert!
age :: Person -> Int -- Schon definiert!
```

Curried functions | Partially applied funcions

```haskell
mulwith :: a -> a -> a
mulwith x y = x * y

mulwith 2 -- was passiert?
```

Curried functions | Partially applied funcions

```haskell
mulwith :: Int -> Int -> Int
mulwith x y = x * y

calc :: Int -> (Int -> Int) -> Int
calc x f = f $ x

main :: IO ()
main = do putStrLn $ show $ calc 5 (mulwith 5)
-- 25

-- Actually 1,1M
```

Curried functions | Partially applied funcions

```haskell
applytwice :: (a -> a) -> a -> a
applytwice f x = f (f x)

applytwice (+ 2) 5
-- 9
```

Resources

`learnyouahaskell.com`

Live Haskell

$$sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} * x^{2k+1} \tag{7}$$

Live Haskell

$$sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} * x^{2k+1} \tag{8}$$

$$cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} * x^{2k} \tag{9}$$