

Haskell - Eine Einführung

Matthias Beyer

Furtwangen University

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

8. Mai 2014

- 1 Facts
- 2 Eine Funktionale Sprache
- 3 Haskell
- 4 Höhere Funktionen
- 5 Pattern Matching
- 6 Guards
- 7 Types
- 8 Partially applied functions
- 9 Resources
- 10 Live hacked

Was, wie, wo, wann, wer?

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

Wikipedia

- APL
- LISP
- Miranda
- ML

Wikipedia

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

Wikipedia

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

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

Wikipedia

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

Wikipedia


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

$$f(x, y) = x \quad (1)$$

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

$$f(x) = x * x = x^2 \quad (2)$$

Typen von Funktionen

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

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

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

- Funktionen bekommen Funktionen als Parameter

Problem: Array mit Ints, alle im Quadrat.

$$f(a, g(x)) = \forall z \in a : g(z) \quad (3)$$

Lösung in C mit Funktion höherer Ord.

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

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

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

Note: Rekursiv!

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

Pattern matching

Pattern Matching

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


Pattern Matching

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

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

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

Pattern Matching

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

1 (1, 2, 3) — A tupel of three

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

$$f(t) = t_1, |t| = 3 \quad (6)$$

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"
```

```
bmiTell :: (RealFloat a) => a -> String
```

```
2 bmiTell weight height
```

```
3     | (weight / height ^ 2) <= 18.5 = "You're underwe
```

```
4     | (weight / height ^ 2) <= 25.0 = "You're normal"
```

```
5     | (weight / height ^ 2) <= 30.0 = "You're fat "
```

```
6     | otherwise = "You crashed the system"
```

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

Also auch ungefähr:

```
data Int = -2147483648 | ... | -1 | 0 | 1 | ... | 2147483647
```

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

Surface berechnen:

```
2 sf :: Shape -> Float  
3  
4 sf (Circle _ _ r) = pi * r ^ 2  
5  
6 sf (Rectangle x1 y1 x2 y2) =  
    (abs $ x2-x1)*(abs $ y2-y1)
```


Surface berechnen:

```
sf $ Circle 10 20 10 — 314.15927  
2 sf $ Rectangle 0 0 100 100 — 10000.0
```

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]
```

Wir verbessern unsern Typ:

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

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)
```

Und nutzen sie:

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

Typdefinition in schön:

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

Haskell hilft:

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

1 `mulwith :: a -> a -> a`

`mulwith x y = x * y`

3 `mulwith 2 — was passiert?`


```
2 mulwith :: Int -> Int -> Int
```

```
mulwith x y = x * y
```

```
4 calc :: Int -> (Int -> Int) -> Int
```

```
calc x f = f $ x
```

```
6 main :: IO ()
```

```
8 main = do putStrLn $ show $ calc 5 (mulwith 5)
```

```
— 25
```

```
— Actually 1,1M
```

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

```
3 applytwice (+ 2) 5
```

```
5 — 9
```

`learnyouahaskell.com`

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

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

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