Programmierung und Modellierung mit Haskell

Teil 6: Funktionen höherer Ordnung

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

14. Mai 2018

Teil 6: Funktionen höherer Ordnung

1. Funktionen höherer Ordnung
   - Funktionen als Rückgabewerte
   - Funktionen als Argumente
2. Elementare HO-Funktionen
   - ... auf Listen
   - ... allgemein
   - Typordnung

Funktionstypen

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht.

Klammerkonvention

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

`Int -> Int -> Int` wird gelesen als `Int -> (Int -> Int)`

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

`bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Int -> Int`
Funktionen sind also normale *Werte* in einer funktionalen Sprache!

Funktionstypen

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht.

Klammerkonvention

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

`Int -> Int -> Int` wird gelesen als `Int -> (Int -> Int)`

Was würde `(Int -> Int) -> Int` bedeuten?

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

`bar 1 8` wird gelesen als `(bar 1) 8`

Das bedeutet: `(bar 1)` ist eine Funktion des Typs `Int -> Int`

Funktionen sind also normale *Werte* in einer funktionalen Sprache!

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> (a -> (a  -> a ))
> :type foo 1
foo 1 :: Num a => a -> (a  -> a)
> :type foo 1 2
foo 1 2 :: Num a => a -> a
> let bar = foo 1 2
bar :: Integer -> Integer
> bar 3
5
```

Achtung: Nicht verwechseln mit *partiellen* Funktionen!

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Es ist natürlich auch möglich, gar keine Argumente anzugeben.

Beispiel:

```
> let foo x y z = x * y + z
foo :: Num a => a -> a -> a -> a

> let bar = foo
bar :: Integer -> Integer -> Integer -> Integer
```

Bemerkung: Die hier von GHCI durchgeführte Spezialisierung des Typs von `Num a` nach `Integer` ist nicht zwingend. `foo` und `bar` dürfen auch den gleichen Typ haben.

`-XMonomorphismRestriction` abschalten

Partielle Funktionsanwendung

Bei **partieller Funktionsanwendung** werden einer Funktion nicht alle Argumente übergeben. Das Ergebnis ist wieder eine Funktion.

Das Ergebnis ist insbesondere ein Wert.
Werte mit Funktionstyp sind Werte wie alle anderen auch.

Beispiel: Wir können sie in Listen packen.

```
> let myOps = [max,min,(+),(-),(*),(/)]
myOps :: [Double -> Double -> Double]

> head myOps 33 44
44.0

> head (drop 2 myOps) 33 44
77.0
```

Funktionen als Rückgabewert

```
> let divBy  x = \y -> (y/x)
divBy :: Fractional a => a -> a -> a

> let drittel = divBy 3
drittel :: Double -> Double
> drittel 9
3.0
> drittel 12
4.0
```

- Werte mit Funktionstyp (also z.B. das Ergebnis von `divBy 3`) wird **Funktionsabschluss** genannt **(engl.: closure)** und besteht aus Umgebung und Funktionsrumpf.
- Die **Umgebung** erklärt/schließt freie Variablen im Funktionsrumpf.
- Werte in der Umgebung sind wie immer unveränderlich, d.h. eine Closure kann gefahrlos mehrfach verwendet werden

  ⇒ Referentielle Transparenz, Folie 1.17

Funktionen als Rückgabewert

```
> let divBy  x = \y -> (y/x)
divBy :: Fractional a => a -> a -> a

> let drittel = divBy 3
drittel :: Double -> Double
> drittel 9
3.0
> drittel 12
4.0
```

- Werte mit Funktionstyp (also z.B. das Ergebnis von `divBy 3`) wird **Funktionsabschluss** genannt **(engl.: closure)** und besteht aus Umgebung und Funktionsrumpf.
- Die **Umgebung** erklärt/schließt freie Variablen im Funktionsrumpf.
- Werte in der Umgebung sind wie immer unveränderlich, d.h. eine Closure kann gefahrlos mehrfach verwendet werden
  ⇒ Referentielle Transparenz, Folie 1.17

> Variable `x` kommt im Rumpf von `divBy` frei vor.
> Für den mit `drittel` bezeichneten *Wert* muss sich die Closure merken, dass `x` den Wert `3` hat.

Funktionen als Rückgabewert

```
divBy :: Fractional a => a -> (a -> a)
```

Alle folgenden Funktionsdefinitionen sind äquivalent:

1. ... mit anonymer Funktionsdefinition
   ```
   divBy_v1 x    = \y -> (y/x)
   ```
2. ... mit lokaler Funktionsdefinition mit `let`/`where`
   ```
   divBy_v2 x    = myDivH
     where myDivH y = (y/x)
   ```
3. Partielle Anwendung von Infix-Funktionen mit Klammerung
   ```
   divBy_v3 x    = (/x)
   ```
   ⇒ engl. Sections
4. Vollständige Definition — diese ist ja auch partiell anwendbar!
   ```
   divBy_v4 x y = (y/x)
   ```

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) -> (a -> a)
twice f = \x -> f (f x)

> twice (+3) 4
10

> twice reverse [1..3]
[1,2,3]

> twice ("ha " ++) "hi"
"ha ha hi"
```

Die Funktion `twice` nimmt also eine Funktion und wendet diese zwei Mal an.

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) -> (a -> a)
twice f = \x -> f (f x)

> twice (+3) 4
10
```

Beachte:
Teilausdruck `(+3)` ist eine Funktion des Typs `Int -> Int`, es ist die Funktion "plus 3".

Generell darf man Infix-Operatoren mit einem der beiden Argument in runde Klammern schreiben und erhält die entsprechende Funktion, welche das andere Argument noch erwartet. Stichwort: "Sections"

Die Fu
zwei Mal an.

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) ->  a -> a
twice f = \x -> f (f x)
```

Die Funktion `twice` können wir natürlich auch wieder partiell anwenden, um eine neue Funktion zu erhalten:

```
> twice (twice (++"ha")) "Mua"
"Muahahahaha"
```

oder auch allgemeiner:

```
quad :: (a -> a) ->  a -> a
quad f = twice (twice f)
> quad ('a':) "rgh!"
"aaaargh!"
> quad (quad (+1)) 0
16
```

Funktion als Argumente

Wir können Funktionstypen auch anders klammern:

```
twice :: (a -> a) ->  a -> a
twice f = \x -> f (f x)
```

Die Funktion `twice` können wir natürlich auch wieder partiell anwenden, um eine neue Funktion zu erhalten:

```
> twice (twice (++"ha")) "Mua"
"Muahahahaha"
```

Hinweis: Auch hier haben wir wieder zwei Sections; Teilausdrücke `(++“ha“)` und `(’a’:)` sind beide Funktionen `String -> String`

```
quad :: (a -> a) ->  a -> a
quad f = twice (twice f)
> quad ('a':) "rgh!"
"aaaargh!"
> quad (quad (+1)) 0
16
```

Funktionen als Argumente

Ganzzahliges Maximum einer Funktion in einem Bereich bestimmen

```
fmax :: (Eq a, Enum a, Ord b) => (a -> b) -> (a,a) -> b
fmax f (bmin, bmax)
    | bmin ==  bmax   = f_bmin
    | f_bmin > f_max = f_bmin
    | otherwise       = f_max
  where
    f_max  = fmax f (succ bmin, bmax)
    f_bmin = f bmin

> fmax (\x-> 2*x^3 -30*x^2 +10) (-5,5)
10

> [ 2*x^3 -30*x^2 +10 | x<-[-5..5]]
[-990,-598,-314,-126,-22,10,-18,-94,-206,-342,-490]
```

Verallgemeinerung

Aus einer Liste von Früchten eine Liste von Preisen machen:

```
data Frucht = Frucht {sorte::Sorte, preis,anzahl::Int}
data Sorte  = Apfel | Birne | Banane Double

preise :: [Frucht] -> [Int]
preise []     = []
preise (f:fs) = (preis f):(preise fs)
```

Aus einer Liste von Zahlen eine Liste von Strings machen:

```
euro :: Int -> String
euro p = (show p) ++ "€"

toEuros :: [Int] -> [String]
toEuros []     = []
toEuros (p:ps) = (euro p):(toEuros ps)
```

Frage: Wo ist der Unterschied zwischen `preise` und `toEuros`?

Verallgemeinerung

In beiden Fällen gehen wir rekursiv durch eine komplette Liste durch und wenden auf jedes Element eine Funktion an.

Den vorangegangenen Satz können wir direkt in Code fassen:

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

Damit können wir die beiden vorangegangenen Funktionen leicht formulieren:

```
-- preis :: Frucht -> Int
preise :: [Frucht] -> [Int]
preise  fs = map preis fs

-- euro :: Int -> String
toEuros :: [Int] -> [String]
toEuros ps = map euro ps
```

Map

Standardbibliothek definiert polymorphe `map` Funktion:

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

Beispiele:

```
> map (compare 5) [3..7]
[GT,GT,EQ,LT,LT]

> map (replicate 3) [1..4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

Allgemein: $\texttt{map}\ f\ [a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]$

Filter

Filtern von Listen ohne List-Comprehension, d.h. Entfernung von allen Elementen, welche einem Prädikat nicht genügen:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []     = []
filter  pred (x:xs)
  | pred x          = x : filter pred xs
  | otherwise       = filter pred xs
```

```
> filter (>= 5) [3..7]
[5,6,7]

> filter (`elem` ['A'..'Z']) "Obst Kaufen!"
"OK"
```

Map & Filter

Map wendet Funktion auf jedes Element einer List an

- Ergebnis ist Liste *anderen Typs*
- Länge bleibt exakt *gleich*

Filter wendet Prädikat (Funktion mit Ergebnistyp `Bool`) auf jedes Element einer Liste an, und entfernt alle Elemente für die dabei `False` herauskommt

- Ergebnis ist Liste *gleichen* Typs
- Länge kann *kleiner* werden

`map` und `filter` sind wesentliche Bestandteile von List-Comprehensions:

```
foo  f p xs = map f (filter p xs))
foo' f p xs = [f x | x <- xs, p x]
```

List-Comprehensions können geringfügig mehr (z.B. refutable patterns, mehrfache Generatoren), aber Funktionen höherer Ordnung lassen sich oft schöner kombinieren.

zipWith

`zipWith` verschmilzt zwei Listen mithilfe einer 2-stelligen Funktion, bei ungleicher Länge wird der Rest der längeren Liste ignoriert:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _    _      _   = []
```

Beispiele:

```
> zipWith (,) [1..7] [20,19..0]
[(1,20),(2,19),(3,18),(4,17),(5,16),(6,15),(7,14)]

> zipWith (+) [10..20] [1..100]
[11,13,15,17,19,21,23,25,27,29,31]

> let chg x y = if x then y else negate y
> zipWith chg [True,False,False,True,False] [1..11]
[1,-2,-3,4,-5]
```

Falten auf Links

`foldl` faltet eine Liste mit einer binären Funktion zusammen. Die Klammerung "lehnt" dabei nach links:

$$
\begin{aligned}
\texttt{foldl}\ f\ \ b\ [a_1, \ldots, a_n] &= f\ \ (\cdots (f\ \ (f\ \ b\ a_1)\ a_2) \cdots)\ a_n \\
&= (\cdots ((b\ `f`\ a_1)\ `f`\ a_2) \cdots)\ `f`\ a_n
\end{aligned}
$$

Haskell code:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc    []   = acc
foldl f acc (h:t) = foldl f (f acc h) t
```

Beispiel

```
> let sum      = foldl (+) 0
> sum [1..9]
45
```

```
product = foldl (*) 1      :: [Double] -> Double
and     = foldl (&&) True :: [Bool] -> Bool
```

Falten auf rechts

`foldr` faltet eine Liste mit einer binärer Funktion zusammen.
Die Klammerung "lehnt" dabei aber nach rechts:

$$
\begin{aligned}
\texttt{foldr}\ f\ b\ [a_1,\ldots,a_n] &= f\ a_1\ (f\ a_2(\cdots(f\ a_n\ b)\cdots)\\
&= a_1\ `f`\ (a_2\ `f`\ (\cdots(a_n\ `f`\ b)\cdots)
\end{aligned}
$$

Haskell code:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc    []   = acc
foldr f acc (h:t)  = f h (foldr f acc t)
```

Beispiele

```
sum      = foldr (+)
and      = foldr (&&)
length   = foldr (\_ n -> succ n) 0
map f    = foldr (\x acc -> f x : acc) []
```

Foldl vs. Foldr

Wann verwendet man `foldl` und wann `foldr`?

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

Das hängt von der Situation ab:

- Wenn die binäre Funktion nicht *assoziativ* ist, dann hat man meist gar keine Wahl
- `foldl` ist endrekursiv, und damit effizienter, wenn immer die gesamte Liste gefaltet werden muss.
- `foldr` ist oft besser, wenn die binäre Funktion nicht immer beide Argumente inspiziert (z.B. `&&`):
  In diesem Fall kann `foldr` abbrechen, ohne die gesamte Liste zu bearbeiten und ist damit effizienter.

Varianten

Es gibt einige Varianten von `foldl` und `foldr`:

z.B. Varianten, welche das Startelement aus der Liste nehmen:

```
foldr1  :: (a -> a -> a) ->      [a] -> a
foldl1  :: (a -> a -> a) ->      [a] -> a
```

z.B. Varianten, welche auch die Zwischenergebnisse ausgeben:

```
scanl   :: (b -> a -> b) -> b -> [a] -> [b]
scanr   :: (a -> b -> b) -> b -> [a] -> [b]
scanl1  :: (a -> a -> a) ->      [a] -> [a]
scanr1  :: (a -> a -> a) ->      [a] -> [a]
```

Achtung: alle ...`1` Varianten liefern auf leeren Listen einen Fehler!

FLIP

Die simple Funktion `flip` vertauscht die Reihenfolge der Argumente einer zweistelligen Funktion:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y  =  f y x
```

Beispiel:

```
subtract :: (Num a) => a -> a -> a
subtract = flip (-)
```

`subtract` ist nützlich, da `(-1)` ausnahmsweise nicht die Funktion `(\x -> (x-1))` darstellt, sondern die Konstante -1 ist.

Currying

Wegen partieller Applikation ist es nahezu unerheblich, ob eine Funktion ein Paar von Argumenten oder zwei Argumente nacheinander erhält. Wir können beliebig wechseln:

```
curry :: ((a,b) -> c)  ->  (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c)  ->  ((a,b) -> c)
uncurry f (x,y) = f x y

> curry snd 42 69
69
> uncurry (+) (3,4)
7
```

Funktionstypen nach dem Muster `A -> (B -> C)` nennt man auch "curried function"

Currying

Wegen partieller Applikation ist es nahezu unerheblich, ob eine Funktion ein Paar von Argumenten oder zwei Argumente nacheinander erhält. Wir können beliebig wechseln:

```
curry :: ((a,b) -> c)  ->   a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c)  ->   (a,b) -> c
uncurry f (x,y) = f x y

> curry snd 42 69
69
> uncurry (+) (3,4)
7
```

Funktionstypen nach dem Muster `A -> (B -> C)` nennt man auch “curried function”

Currying

Das Prinzip lässt sich natürlich auf Funktionen beliebiger Stelligkeit anwenden, hier z.B. für dreistellige Funktionen:

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

uncurry3 :: (a -> b -> c -> d) -> (a,b,c) -> d
uncurry3 f (x,y,z) = f x y z
```

Wenn wir von einer *n*-stelligen Funktion reden, unterscheiden wir in unserer Sprechweise gar nicht mehr zwischen diesen Varianten.

Benannt nach Haskell Curry (1900–82, amerikanischer Logiker); ausgearbeitet von Moses Schönfinkel (1889–1942, russischer Logiker); aufbauend auf Arbeiten von Gottlob Frege (1848–26, deutscher Logiker).

Pointfree Style

Currying erlaubt es uns oft, Funktionen als Komposition anderer Funktionen zu beschreiben, ohne dabei die Argumente explizit zu erwähnen:

```
sum :: Num a => [a] -> a
sum xs = foldr (+) 0 xs
```

können wir einfacher schreiben als

```
sum     = foldr (+) 0
```

Argumente am Ende können wir einfach weglassen,
da der Typ `a -> b -> c` ja identisch zu `a -> (b -> c)` ist!
`(\x y-> foobar 42 x y)` wird verwendet wie `(foobar 42)`

Dieser Programmierstil wird als **“Pointfree”** (Punktfrei) bezeichnet, und ist manchmal besser lesbar

Begriffsbezeichnung in der Kategorientheorie begründet

Hintereinanderausführung

Mit der Infix Funktion `(.)` können wir zwei Funktionen miteinander verketten:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

```
> ( (4+) . (10*) . succ . max 1 . \n -> n*n+1 ) 1
34
```

Damit können wir auch gleich eine Liste von Funktionen der Reihe nach ausführen:

```
compose :: [a -> a] -> a -> a
compose = foldr (.) id
```

```
> compose [(4+), (10*), succ, max 1, \n -> n*n+1] 1
34
```

Pointfree dank Punkt

Funktionskomposition ermöglicht oft den Punktfreien Stil:

```
bar x = ceiling (negate (cos x))
bar'  = ceiling . negate . cos

foo' :: (Int->b) -> (Int->Bool) -> [Int] -> [b]
foo'  f p xs = map f (filter p (filter (>=0) xs))
foo'' f p    = map f . filter p . filter (>=0)
```

Man sieht so oft besser, wie eine Funktion aus anderen Funktinen zusammensetzt wird.

Allgemein:

```
(\x -> f5(f4(f3(f2(f1 x))))) == f5.f4.f3.f2.f1
```

Ironischerweise nutzt der pointfree-style viele Punkt-Operatoren

Pointless Style

```
foo :: (Int->b) -> (Int->Bool) -> [Int] -> [b]
foo'  f p xs = map f (filter p (filter (>=0) xs))
foo'' f p    = map f . filter p . filter (>=0)
```

Längere Kompositionsketten werden schnell unlesbar; dann ist es oft besser, den Zwischenergebnissen sprechende Namen zu geben:

```
foo''' f p xs_input =
    let xs_positives  = filter (>=0) xs_input
        xs_p_filtered = filter p xs_positives
        xs_mapped     = map f xs_p_filtered
    in xs_mapped
```

Vorsicht: Man darf nicht überall den gleichen Bezeichner verwenden, weil es dann als rekursive Definition verstanden wird und damit nicht mehr terminiert!

$ Funktion

Was macht diese Infix-Funktion?

```
($)    :: (a -> b) -> a -> b
f $ x =  f x
```

$ Funktion

Was macht diese Infix-Funktion?

```
($)    :: (a -> b) -> a -> b
f $ x =  f x
```

Antwort: Funktionsanwendung / Klammern sparen!

Im Gegensatz zu dem Leerzeichen als Funktionsanwendung, hat `$` eine sehr niedrige Präzedenz (Bindet schwach).

Merke: `$` ersetzt Klammer, welche so spät wie möglich schliesst

Beispiel:

```
sum (filter (> 10) (map (^2) [1..10]))
```

ist gleichwertig zu

```
sum $ filter (>10) $ map (^2) [1..10]
```

$ Funktion

Was macht diese Infix-Funktion?

```
infixr 0  $

($)    :: (a -> b) -> a -> b
f $ x =  f x
```

Weiterhin erlaubt `$` auch die Verwendung der Funktionsanwendung selbst als Funktion:

```
> map ($ 3) [(4+), (10*), succ, max 1, \n -> n*n+1, id]
[7,30,4,3,10,3]
```

Hinweis:
`$` keine eingebaute Syntax, sondern gewöhnliche Infix Funktion!

$ Funktion

Damit haben wir noch eine Variante:

```
foo'''' f p xs = map f $ filter p $ filter (>=0) xs

foo'    f p xs = map f (filter p (filter (>=0) xs))

foo''   f p    = map f . filter p . filter (>=0)

foo'''  f p xs_input =
    let xs_positives = filter (>=0) xs_input
        xs_filtered  = filter p xs_positives
        xs_mapped    = map f xs_filtered
    in xs_mapped

foo     f p xs = [f x | x <- xs, x >= 0, p x]
```

⇒ Was jeweils am “schönsten” ist, hängt von der Situation ab.

Ordnung

Der Name “Funktion höherer Ordnung” stammt von der üblichen Definition der **Typordnung** ab:

$$\mathrm{ord}(T) = \begin{cases} 0 & \text{falls } T \text{ ein Basistyp ist} \\ \max\big(1+\mathrm{ord}(A), \mathrm{ord}(B)\big) & \text{falls } T \equiv A \to B \end{cases}$$

Beispiele:

$$\mathrm{ord}\Big(\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}\Big) = 1$$

$$\mathrm{ord}\Big(\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}\Big) = 2$$

$$\mathrm{ord}\Big((\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int})) \to \mathtt{Int}\Big) = 2$$

$$\mathrm{ord}\Big(((\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}) \to \mathtt{Int}\Big) = 3$$

Zusammenfassung

- Funktionen sind ganz normale Werte in Haskell
- Funktionen höherer Ordnung nehmen Funktionen als Argumente (und können Funktionen zurück liefern)
- Funktionen höherer Ordnung abstrahieren auf einfache Weise häufig verwendete Berechnungsverfahren;
  viele wichtige in Standardbibliothek verfügbar
- Funktionsanwendung darf partiell sein;
  partielle Funktionsanwendung liefert eine Funktion
- Die durch Funktionen höherer Ordnung gewonnene Modularität erlaubt sehr viele Kombinationsmöglichkeiten

Evolution of a Haskell Programmer

Funktionen höherer Ordnung sind ein mächtiges Werkzeug, welches sehr vielfältig benutzt werden kann. Welches Werkzeug für welchen Fall geeignet ist, muss von Fall zu Fall unterschieden werden.

Die humoristische Webseite Evolution of a Haskell Programmer fasst dies ganz gut zusammen. Auf welcher Stufe bist Du?

Wichtig ist, dass der Code lesbar bleibt, denn:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> *C.A.R. Hoare*
>
> *1980 ACM Turing Award Lecture*