Einführung in Haskell

Axel Stronzik

21. April 2008

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Vorteile funktionaler Programmiersprachen

- hohes Abstraktionsniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte, dies führt zu höherer Verständlichkeit und besseren Möglichkeiten zur Code-Optimierung
- Programmierung über Eigenschaften, nicht über den zeitlichen Ablauf
- kompakterer Source-Code (kürzere Entwicklungszeiten, lesbarere Programme, bessere Wartbarkeit)
- modularer Programmaufbau, Polymorphismus, Funktionen höherer Ordnung, damit auch eine hohe Wiederverwertbarkeit des Codes
- einfachere Korrektheitsbeweise

Strukturen in funktionalen Programmiersprachen

- Variablen entsprechen unbekannten Werten (nicht Speicherzellen!)
- Speicher ist nicht explizit verwendbar, sondern wird automatisch alloziert und freigegeben (Garbage Collection)
- Programme entsprechen Mengen von Funktions- und Typdefinitionen
- Programmablauf entspricht einer Reduktion von Ausdrücken (nicht einer Sequenz von Anweisungen)

Haskell

- **Unix:** GHCI (auf den Suns vorinstalliert unter: /home/haskell/bin/ghci)
- **Windows:** Hugs (http://www.haskell.org/hugs)

Funktionen

Die Definition einer Funktion in Haskell:

$$f\ x_1 \ldots x_n = e$$

Mögliche Ausdrücke die man nutzen kann:

- Zahlen $(3, 3.1415\ldots)$
- Basisoperationen $(3 + 4, 6 * 7)$
- formale Parameter (Variablen)
- Funktionsanwendung $((f\ e_1 \ldots e_n))$ mit einer Funktion f und Ausdrücken e_i
- bedingte Ausdrücke `(if b then e1 else e2)`

Beispielfunktionen

- Quadratfunktion:

```
square x = x * x
```

Beispielfunktionen

- Quadratfunktion:

```
square x = x * x
```

- Minimumsfunktion:

```
min x y = if x <= y then x else y
```

Beispielfunktionen

- Fakultätsfunktion:

$$n! = \begin{cases} 1 & \text{falls} n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

Beispielfunktionen

- Fakultätsfunktion:

$$n! = \begin{cases} 1 & \text{falls} n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

- Wird zu:

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

Beispielfunktionen

- Fakultätsfunktion:

$$n! = \begin{cases} 1 & \text{falls} n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

- Wird zu:

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

- Fibonacci:

```
fib1 n = if n == 0
           then 0
           else if n == 1
                  then 1
                  else fib1(n - 1) + fib1(n - 2)
```

Akkumulatortechnik

Verbesserung

```
fib' fibn fibnp1 n =
      if n == 0
         then fibn
         else fib' fibnp1 (fibn + fibnp1) (n - 1)

fib2 n = fib' 0 1 n
```

Lokale Definitionen

- Unötige toplevel Funktion fib'.

Lokale Definitionen

- Unötige toplevel Funktion fib'.

- Besser lokal definieren:

```
fib n = fib' 0 1 n
          where fib' fibn fibnp1 n = . . .
```

Lokale Definitionen

- Unötige toplevel Funktion fib'.

- Besser lokal definieren:

```
fib n = fib' 0 1 n
            where fib' fibn fibnp1 n = . . .
```

- Oder mit let:

```
fib n = let fib' fibn fibnp1 n = . . .
        in fib' 0 1 n
```

Off-Side-Rule

Sichtbarkeiten

```
f x = e
    where g y = {Rumpf von g}
                {Rumpf von g geht weiter}
                {Rumpf von g geht weiter}
          h z = {Rumpf von h}
                {Rumpf von h geht weiter}
                {Rumpf von h geht weiter}
  k x y = . . .
```

Off-Side-Rule

Sichtbarkeiten

- ```
  f x = e
      where g y = {Rumpf von g}
                  {Rumpf von g geht weiter}
                  {Rumpf von g geht weiter}
            h z = {Rumpf von h}
                  {Rumpf von h geht weiter}
                  {Rumpf von h geht weiter}
    k x y = . . .
  ```
- Im Rumpf von g: Variablen x, y, Funktionen f, g, h, k
  ```

Off-Side-Rule

Sichtbarkeiten

- ```
  f x = e
      where g y = {Rumpf von g}
                  {Rumpf von g geht weiter}
                  {Rumpf von g geht weiter}
            h z = {Rumpf von h}
                  {Rumpf von h geht weiter}
                  {Rumpf von h geht weiter}
    k x y = . . .
  ```
- Im Rumpf von g: Variablen x, y, Funktionen f, g, h, k
- Im Rumpf von h: Variablen x, z, Funktionen f, g, h, k
  ```

Off-Side-Rule

Sichtbarkeiten

- ```
  f x = e
      where g y = {Rumpf von g}
                  {Rumpf von g geht weiter}
                  {Rumpf von g geht weiter}
            h z = {Rumpf von h}
                  {Rumpf von h geht weiter}
                  {Rumpf von h geht weiter}
    k x y = . . .
  ```
- Im Rumpf von g: Variablen x, y, Funktionen f, g, h, k
- Im Rumpf von h: Variablen x, z, Funktionen f, g, h, k
- Im Rumpf von k: Variablen x, y, Funktionen f, k
  ```

Basisdatentypen

Int Werte von $-2^{16}+1$ bis $2^{16}-1$
(**Integer**, beliebig kleine/große Werte, durch Speicher beschränkt)
Operationen: $+$, `-`, `*`, `div`, `mod`...
Vergleiche: $<=, >=, <, >, /=, ==$

Bool **Boolesche Werte:** `True, False`
Operationen: &&, ||, $==$, $/=$, `not`

Float **Fließkommazahlen**, geschrieben $0.3, -1.5e-2$
Operationen wie bei **Int** mit / statt `div` und ohne `mod`

Char **Zeichen (ASCII)**, mit Werten: `'a', '\n', '\NUL', '\214'`
Operationen: `chr, ord`

Typannotationen

Alle Werte in Haskell haben einen **Typ**. Dieser kann mittels **:: annotiert** werden.

Beispiele:

- 3 :: **Integer**
- 3 :: **Int**
- (3 == 4) || **True** :: **Bool**
- square :: **Int** -> **Int**
- square x = (x :: **Int**) * (x :: **Int**) :: **Int**
- min :: **Int** -> **Int** -> **Int**

Algebraische Datentypen

Eigenen τ Datentypen definieren mit dem Befehl `data`:

$$\texttt{data}\ \tau = c_1\ \tau_{11}\ \ldots\ \tau_{1n_1} \mid \ldots \mid c_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}$$

`wobei:`

- τ `der neu definierte Typ ist`
- $c_1, \ldots\ c_k$ `definierte Konstruktoren sind`
- τ_{i1} `bis` τ_{in_i}`, die Argumenttypen des Kontruktors` c_i `sind, also` $c_i :: \tau_{i1} -> \ldots\ \tau_{in_i} -> \tau$

Beispiele

- ```
  data Color = Red | Blue | Yellow
  Red :: Color
  ```
  ```

Beispiele

- `data Color = Red | Blue | Yellow`
  `Red :: Color`
- data Complex = Complex Float Float
  Complex 3.0 4.2 :: Complex

  Anwendung:

```
addC :: Complex -> Complex -> Complex
addC (Complex r1 i1)
     (Complex r2 i2) =
         Complex (r1 + r2) (i1 + i2)
```

Beispiele

```
data Color = Red | Blue | Yellow
```

Auswertung von `Red` $\rightsquigarrow$ `Error`

Beispiele

```
data Color = Red | Blue | Yellow
  deriving (Eq, Show)
```

Auswertung von `Red` $\rightsquigarrow$ `Error`

Beispiele

- data List = Nil | Cons Int List

Beispiele

- data List = Nil | Cons Int List

  Anwendung:

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Beispiele

- data List = Nil | Cons Int List

  Anwendung:

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

- Vordefinierte Listen in Haskell:

```
data [Int] = [] | Int : [Int]
```

Beispiele

- data List = Nil | Cons Int List

  Anwendung:

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

- Vordefinierte Listen in Haskell:

```
data [Int] = [] | Int : [Int]
```

  Anwendung: Berechnung der Länge einer Liste:

Beispiele

- data List = Nil | Cons Int List

  Anwendung:

```
append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

- Vordefinierte Listen in Haskell:

```
data [Int] = [] | Int : [Int]
```

  Anwendung: Berechnung der Länge einer Liste:

```
length :: [Int] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Polymorphismus

Bisher Jede Definition ist auf einen Typ festgelegt. Man muss für jeden Typ eigene Funktion/Konstruktor schreiben.

Polymorphismus

Bisher Jede Definition ist auf einen Typ festgelegt. Man muss für jeden Typ eigene Funktion/Konstruktor schreiben.

Verbesserung Polymorphismus. Einführen von Typvariablen.

Polymorphismus

Bisher Jede Definition ist auf einen Typ festgelegt. Man muss für jeden Typ eigene Funktion/Konstruktor schreiben.

Verbesserung Polymorphismus. Einführen von Typvariablen.

Anwendung

```
data [a] = [] | a : [a]  -- so nicht implementiert

length :: [a] -> Int
```

Beispiele

- ```
  data Maybe a = Nothing | Just a

  isNothing :: Maybe a -> Bool
  isNothing Nothing = True
  isNothing (Just _) = False
  ```

- ```
  data Tree a = Node (Tree a) a (Tree a) | Empty

  height :: Tree a -> Int
  height Empty = 0
  height (Node tl _ tr) = 1 + (max (height tl)
                                   (height tr))
  ```

Beispiele

- **Strings** sind in Haskell als Typsynonym für Listen von Zeichen definiert:
  `type String = [Char]`

- **Vereinigung** zweier Typen:
  `data Either a b = Left a | Right b`

- **Tupel** mit Mixfixnotation:
  ```
  data (,) a b = (,) a b -- oder (a, b)
  data (,,) a b c = (,,) a b c -- oder (a, b, c)
  ```

Pattern

- **x** (Variable) passt immer, x wird an aktuellen Ausdruck gebunden
- _ (Wildcart) passt immer, keine Bindung
- $c\ pat_1\ \ldots\ pat_k$ (c k-stelliger Konstruktor und $pat_1\ \ldots\ pat_k$ Pattern) passt, falls aktueller Wert $c\ e_1\ \ldots\ e_k$ ist, und alle pat_i auf alle e_i passt.

Pattern

- **x** (Variable) passt immer, x wird an aktuellen Ausdruck gebunden
- _ (Wildcart) passt immer, keine Bindung
- $c\ pat_1\ \ldots\ pat_k$ (c k-stelliger Konstruktor und $pat_1\ \ldots\ pat_k$ Pattern) passt, falls aktueller Wert $c\ e_1\ \ldots\ e_k$ ist, und alle pat_i auf alle e_i passt.

- `x@pat` ("as pattern") passt, falls `pat` passt, zusätzliche Bindung von x an den aktuellen Ausdruck
- `n + k` Passt auf alle Zahlen größer gleich `k`, wobei `n` an aktuellen Wert minus `k` gebunden wird.

Nicht erlaubte Pattern

Nicht erlaubt: gleiche Variable in einem Pattern

```
eq :: a -> a -> Bool
eq x x = True      -- nicht erlaubt
eq _ _ = False     -- erlaubt
```

Der case Ausdruck

Oft ist Patternmatching in bedingten Ausdrücken nützlich. Dazu gibt es den `case` Ausdruck:

```
case e of
    pat1 -> e1
    ...
    patn -> en
```

Wichtig: Die Pattern für Ausdruck `e` müssen denselben Typ haben!

Guards

Guards erlauben es zusätzliche Bedingungen an die "linke" Regelseite (Pattern) zu knüpfen.

Beispiele:

```
fac n | n == 0    = 1
      | otherwise = n * fac (n - 1)
```

```
take :: Int -> [a] -> [a]
take n _ | n <= 0     = []
take _ []             = []
take (n + 1) (x : xs) = x : take n xs
```

Guards sind bei jedem Pattern Matching erlaubt, also auch bei `case`-, `let` und `where`-Ausdrücken.

Funktionen höherer Ordnung

- **Idee:** Funktionen sind in funktionalen Programmiersprachen "Bürger erster Klasse", d.h. können wie alle anderen Werte überall verwendet werden. Anwendungen sind unter anderem generische Programmierung und Programmschemata (Kontrollstrukturen).
- **Vorteile:** Höhere Wiederverwendbarkeit und große Modularität.

Funktionen höherer Ordnung

- **Idee:** Funktionen sind in funktionalen Programmiersprachen "Bürger erster Klasse", d.h. können wie alle anderen Werte überall verwendet werden. Anwendungen sind unter anderem generische Programmierung und Programmschemata (Kontrollstrukturen).
- **Vorteile:** Höhere Wiederverwendbarkeit und große Modularität.
- **Beispiel:**

```
ableitung :: (Float -> Float) -> (Float -> Float)
```

Funktionen höherer Ordnung

- **Idee:** Funktionen sind in funktionalen Programmiersprachen "Bürger erster Klasse", d.h. können wie alle anderen Werte überall verwendet werden. Anwendungen sind unter anderem generische Programmierung und Programmschemata (Kontrollstrukturen).
- **Vorteile:** Höhere Wiederverwendbarkeit und große Modularität.
- **Beispiel:**

```
ableitung :: (Float -> Float) -> (Float -> Float)
```

Numerische Berechnung: $f'(x) = \lim_{dx \to 0} [\frac{f(x+dx)-f(x)}{dx}]$

In Haskell:

```
dx = 0.0001
ableitung f = f'
        where f' x = (f (x + dx) - f x) / dx
```

Anonyme Funktionen

Um Namensraum zu schonen, können "kleine" Funktionen als anonyme Funktionen definieren.

```
ableitung f = \x -> (f (x + dx) - f x) / dx
```

`'ableitung'` hat hier einen funktionalen Wert.

Partielle Applikation

Betrachte:

```
add :: Int -> Int -> Int
add x y = x + y
```

Partielle Applikation

Betrachte:

```
add :: Int -> Int -> Int
add x y = x + y
```

Mögliche andere Definitionen wären:

- `add = \x y -> x + y`
- `add x = \y -> x + y`

Partielle Applikation

Partielle Applikation ist Typkorrekt, da '$->$' *rechtsassoziativ* ist. Es gilt also:

$$a -> b -> c = a -> (b -> c)$$

$$a -> b -> c \neq (a -> b) -> c$$

Generische Programmierung

```
filter p [] = []
filter p (x : xs) | p x         = x : filter p xs
                  | otherwise = filter p xs
```

Generische Programmierung

- ```
  filter :: (a -> Bool) -> [a] -> [a]

  filter p [] = []
  filter p (x : xs) | p x       = x : filter p xs
                    | otherwise = filter p xs
  ```
  ```

Generische Programmierung

- ```
  filter :: (a -> Bool) -> [a] -> [a]

  filter p [] = []
  filter p (x : xs) | p x        = x : filter p xs
                    | otherwise = filter p xs


  map f [] = []
  map f (x : xs) = f x : map f xs
  ```
  ```

Generische Programmierung

- ```
  filter :: (a -> Bool) -> [a] -> [a]

  filter p [] = []
  filter p (x : xs) | p x       = x : filter p xs
                    | otherwise = filter p xs
  ```
- ```
  map :: (a -> b) -> [a] -> [b]

  map f [] = []
  map f (x : xs) = f x : map f xs
  ```

Generische Programmierung

- ```
  filter :: (a -> Bool) -> [a] -> [a]

  filter p [] = []
  filter p (x : xs) | p x       = x : filter p xs
                    | otherwise = filter p xs
  ```
- ```
  map :: (a -> b) -> [a] -> [b]

  map f [] = []
  map f (x : xs) = f x : map f xs
  ```

```
foldr f e [] = e
foldr f e (x : xs) = f x (foldr f e xs)
```

Generische Programmierung

- ```
  filter :: (a -> Bool) -> [a] -> [a]

  filter p [] = []
  filter p (x : xs) | p x        = x : filter p xs
                    | otherwise = filter p xs
  ```
- ```
  map :: (a -> b) -> [a] -> [b]

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

  foldr f e [] = e
  foldr f e (x : xs) = f x (foldr f e xs)
  ```
  ```

Weitere Funktionen

- ```
  (.) :: (a -> b) -> ( c -> a ) -> c -> b
  (f . g) x = f (g x)
  ```
- ```
  flip :: (a -> b -> c ) -> b -> a -> c
  flip f x y = f y x
  ```
- ```
  const :: a -> b -> a
  const x _ = x
  ```
  ```

Lazy Evaluation

Betrachte folgendes Haskel-Programm:

```
f x = 1
h = h
```

Mit folgender Anfrage:

```
f h
```

Auswertungsstrategien

Es gibt 2 interessante Auswertungsstrategien:

- *Leftmost-Innermost (LI, strikte Funktionen):* Alle Argumente eines Funktionsaufrufs müssen ausgewertet sein, bevor die Funktion angewendet werden kann.
- *Leftmost-Outermost (LO, nicht-strikte Funktionen):* Die Funktionen werden jeweils vor Argumenten ausgewertet. (Berechnungsvollständig)

unendliche Datenstrukturen

Beispiel:

```
from :: Num a => a -> [a]
from n = n : from (n + 1)
```

Nehme nun die bekannte Funktion `take`:

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take n []         = []
take n (x : xs)   = x : take (n - 1) xs
```

Nun ist z.B. folgendes möglich: `take 3 (from 1)` $\rightsquigarrow$ `[1,2,3]`

Typklassen und Überladung

Betrachte folgende Funktion:

```
elem x [] = False
elem x (y : ys) = x == y || elem x ys
```

Möglicher Typ der Funktion:

```
elem :: a -> [a] -> Bool
```

Typklassen und Überladung

Betrachte folgende Funktion:

```
elem x [] = False
elem x (y : ys) = x == y || elem x ys
```

Möglicher Typ der Funktion:

```
elem :: a -> [a] -> Bool
```

Wo liegt das Problem?

Typklassen und Überladung

Der Typ ist zu Allgemein, da a auch eine Funktion repräsentieren kann.
Hier kann die Gleichheit nicht entschieden werden.
Abhilfe schafft eine Einschränkung auf Typen, für die die Gleichheit definiert ist:

```
elem :: Eq a => a -> [a] -> Bool
```

Beispiel

Gleichheit für Bäume implementieren:

```
data Tree = Empty | Node Tree Int Tree

instance Eq Tree where
    Empty == Empty = True
    (Node tl1 n1 tr1) == (Node tl2 n2 tr2) =
          tl1 == tl2 && n1 == n2 && tr1 == tr2
          t1 /= t2 = not (t1 == t2)
```

Totale Ordnung

Erweiterung der **Eq** Klasse zur totalen Ordnung:

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<) , (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a

data Ordering = LT | EQ | GT
```

Klassen

Weitere vordefinierte Klassen:

- **Num** zum Rechnen mit Zahlen (definiert `(+)`, `(-)`, `(*)`, `abs` etc.)
- **Show** zum Umwandeln in Strings:
  `show :: Show a => a -> String`
- **Read** zum Konstruieren aus Strings

Ein-/Ausgabe

Wichtig bei Ein-/Ausgabe ist die Reihenfolge dieser Aktionen. In Haskell realisiert durch den abstrakten Datentyp `IO ()`, z.B.:

```
putChar :: Char -> IO ()
```

Das Zusammensetzen von IO-Aktionen geschieht mittels des *Sequenzoperators*:

```
(>>) :: IO () -> IO () -> IO ()
```

Sequenzialisierung mit Weitergabe durch den *Bind-Operator*:

```
(>>=) : : IO a -> (a -> IO b) -> IO b
```

Beispiel

```
putStr :: String -> IO ()
putStr "" = return ()
putStr (c : cs) = putChar c >> putStr cs

getline :: IO String
getline = getchar >>= \ c ->
    if c == '\n'
        then return ""
        else getline >>= \cs -> return (c : cs)
```

Do-Notation

Die **Do-Notation** ist eine vereinfachte Schreibweise, die statt der Operatoren `(>>=)`, `(>>)` nur Sequenzen verwendet. Mögliche Ausdrücke sind:

- *pat* `<-` *e*
- `let` *pat* `=` *e* `-- ohne in`
- *e*

Beispiel:

```
main = do
   c <- getChar
   putChar c
```

Beachte: Schlüsselwort **do** leitet die Notation ein. Off-Side-Rule auch hier gültig.

Beispiele

getline

in Do-Notation:

```
getline = do
    c <- getChar
    if c == '\n' then return ""
                 else do cs <- getline
                         return (c : cs)
```

Beispiele

Ausgaben aller zwischenergebnisse von `fac`

```
fac :: Int -> IO Int
fac 0 = do
   putStr "1"
   return 1
fac (n + 1) = do
   facN <- fac n
   let facNp1 = (n + 1) * facN
   putStr (' ' : show facNp1)
   return facNp1
```