Haskell for Hackers
. . . or why functional programming matters

Franz Pletz <fpletz@muc.ccc.de>

CCC München

27-06-2009 @ GPN8

Fahrplan

Ablauf

- bei Fragen/Unklarheiten: sofort dazwischenschreien!
- so wenig Mathematik wie möglich
- Prinzipien sprachübergreifend, aber Beschränkung auf Haskell-Syntax
- Hands-On: paralleles Spielen mit dem Interpreter
- Ziele:
  - Einführung in allgemeine Prinzipien der funktionalen Programmierung
  - Haskell-Basis schaffen, um die Sprache selbst weitererforschen zu können

Imperative Programmierung

- hohe Popularität klassicher imperativer Paradigmen
  - folgt aus dem von Neumann Modell
  - Ausführung der Befehle in fixer Reihenfolge
  - interner, globaler State des Programmes definiert Verhalten
- Probleme
  - Debugging durch Abhängigkeit vom State schwierig
  - Verifikation von Programmen komplex
  - Parallelisierung aufwändig und nicht trivial

Fahrplan

Ein paar Worte zu Haskell

- Paradigmen
  - functional
  - non-strict semantics: Funktionsargumente müssen nicht ausgewertet werden
  - lazy evaluation: Auswertung und Berechnung on demand
  - modular
- Typsystem
  - static: Typchecks zur Compilezeit
  - strong: keine automatische Typkonvertierung
  - inferred: Typen können implizit abgeleitet werden
- automatisches Speichermanagement

Higher-Order Functions

- Prinzip: Funktionen können Funktionen als Argumente oder Rückgabewerte haben
- Impliziert First-Class Functions: Erzeugung von Funktionen zur Laufzeit und Behandlung als Objekt
- Auswirkungen
  - Anonyme Funktionen (Lambda, Closures)
  - Currying

Lambda-Ausdrücke

Beispiel

```
let f = \x y -> x + y
```

- Abgeleiteter Typ: `f :: (Num a) => a -> a -> a`
  - `f`: Name
  - `Num`: Typ bzw. Typklasse, numerisch
  - `a`: Platzhalter
  - Deutsch: Funktion `f` nimmt 2 numerische Parameter und gibt einen numerischen zurück
  - Achtung: Immer der selbe Typ a! (z.B. Integer, Fractional)

Lambda-Ausdrücke

Beispiel

```
let f = \x y -> x + y
```

- Abgeleiteter Typ: `f :: (Num a) => a -> a -> a`
  - `f`: Name
  - `Num`: Typ bzw. Typklasse, numerisch
  - `a`: Platzhalter
  - Deutsch: Funktion `f` nimmt 2 numerische Parameter und gibt einen numerischen zurück
  - Achtung: Immer der selbe Typ a! (z.B. Integer, Fractional)

Alternative Definition

```
let f x y = x + y
```

Currying

Definition Funktion f

```
f :: (Num a) => a -> a -> a
let f x y = x + y
```

Beispiel

```
let f5 = f 5
```

- Was ist der Typ? Semantik?

Currying

Definition Funktion f

```
f :: (Num a) => a -> a -> a
let f x y = x + y
```

Beispiel

```
let f5 = f 5
```

- Was ist der Typ? Semantik?
  - `f5 :: Integer -> Integer`
  - initialisiert erstes Argument von f mit 5
  - f benötigt noch ein Argument, neue Funktion!
  - `Num a` wird wegen 5 zu `Integer` spezialisiert

Pure Functions

- Funktionen ohne Seiteneffekte (kein I/O, Syscalls, . . .)
- gleiche Argumente produzieren immer gleiche Rückgabewerte
- keine Veränderung gemeinsamer globaler oder statischer Variablen (State)
- Haskell ist purely functional
- Folgen
  - Order of Evaluation ist egal
  - Call-by-need Evaluation: Lazy Evaluation
  - implizite Thread-Safety: automatisierte Parallelisierbarkeit

Lazy Evaluation: Beispiele

Beispiel 1

```
let f x y = x + 1
f 1 [0..]
```

- `[0..]` erzeugt eine unendlich lange Liste
- jedoch keine Berechnung, da nicht benötigt

Lazy Evaluation: Beispiele

Beispiel 1

```
let f x y = x + 1
f 1 [0..]
```

- `[0..]` erzeugt eine unendlich lange Liste
- jedoch keine Berechnung, da nicht benötigt

Beispiel 2

```
take 2 [0..]
```

- `take n l` nimmt `n` Elemente aus Liste `l`
- unendlich lange Liste wird nur so weit ausgewertet wie benötigt

Pattern Matching

- Datentypen in Haskell werden durch Konstruktoren aufgebaut
- diese fungieren gleichzeitig als „Desktruktoren“ zum zerlegen
- Beispiel: Tupel
  - Tupel: `(1,2)`
  - Zerlegen: `let (a,b) = (1,2) in a + b`
- Beispiel: Listen
  - Liste: `[1,2,3]`
  - mit Cons-Konstruktur: `1:(2:(3:[]))`
  - Zerlegen: `let (x:xs) = [1,2,3] in x`

Datenstrukturen

Syntax

```
data <Typ> <Typ1a = <Konstruktor1> <Typ1a> |
<Konstruktur2> | ... deriving <Typklasse1>, ...
```

Datenstrukturen

Syntax

```
data <Typ> <Typ1a = <Konstruktor1> <Typ1a> |
<Konstruktur2> | ... deriving <Typklasse1>, ...
```

Beispiel 1

```
data Maybe t = Just t | Nothing
```

Datenstrukturen (2)

Beispiel 2: Binärbaum

```
data BTree t = Node (BTree t) t (BTree t) | Leaf
        deriving Show
```

Datenstrukturen (2)

Beispiel 2: Binärbaum

```
data BTree t = Node (BTree t) t (BTree t) | Leaf
        deriving Show
```

Beispiel 3: Natürliche Zahlen

```
data Nat = Zero | Succ Nat
```

Datenstrukturen (2)

Beispiel 2: Binärbaum

```
data BTree t = Node (BTree t) t (BTree t) | Leaf
        deriving Show
```

Beispiel 3: Natürliche Zahlen

```
data Nat = Zero | Succ Nat
```

Beispiel 4: Liste

```
data List a = Nil | Cons a (List a)
```

List Comprehensions

- Erzeugung einer neuen Liste aus einer Operation, einer Menge von Quellisten und einer Menge von Prädikaten
- entspricht der Mengenotation in der Mathematik

Beispiel: Quadrat aller geraden Zahlen von 1 bis 9

$$[\underbrace{x * x}_{\text{Operation}} \mid \underbrace{x}_{\text{Variable}} <- \underbrace{[1..9]}_{\text{Eingabeliste}}, \underbrace{\texttt{x `mod` 2 == 0}}_{\text{Prädikat}}]$$

List Comprehensions: Beispiele

Anwendung einer Funktion für alle Elemente einer Liste

```
let map' f xs = [f x | x <- xs]
```

List Comprehensions: Beispiele

Anwendung einer Funktion für alle Elemente einer Liste

```
let map' f xs = [f x | x <- xs]
```

Pythagoreische Tripel

```
let pythag n = [(a, b, c) | a <- l, b <- l, c <- l,
                a*a + b*b == c*c] where
          where l = [1..n]
```

Rekursion

- Problem: Schleifen sind ein imperatives Konzept
- funktionale Lösung: Rekursion!
- Prinzip: Funktion ruft sich selbst auf um selben Code nochmal auszuführen
- Overhead durch Stack-Management, Compiler können jedoch z.B. Tail-Recursion erkennen und optimieren

Rekursion: Beispiele

Fakultät

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Rekursion: Beispiele

Fakultät

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Anwendung einer Funktion für alle Elemente einer Liste

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

Rekursion: Beispiele

Quicksort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort [y | y <- xs, y < x]) ++
         [x] ++ (qsort [y | y <- xs, y >= x])
```

Fahrplan

Datentypen

- Vordefinierte Typen, z.B.
  - Int, Integer
  - Float
  - Complex
  - Boolean
  - IO t
- Klassen
  - Num
  - Eq
  - Show
  - Ord

Datentypen (2)

Syntax: Neue Datentypen

```
type <Name> = <Typ>
```

Beispiel: Datentyp

```
type FilePath = [String]
```

Monaden

- Prinzip: Es wird ein weiteres Typsystem in ein bereits vorhandenenes hineinbeschrieben
- Grundsätzliche Operationen:
  - Typkonstruktion: `M t`
  - Mappingfunktion: `t -> M t`
    in Haskell `return x`
  - Bindungsoperation: `M t -> (t -> M u) -> M u`
    in Haskell >>= oder do-Ausdrücke

Monaden

- Prinzip: Es wird ein weiteres Typsystem in ein bereits vorhandenenes hineinbeschrieben
- Grundsätzliche Operationen:
  - Typkonstruktion: `M t`
  - Mappingfunktion: `t -> M t`
    in Haskell `return x`
  - Bindungsoperation: `M t -> (t -> M u) -> M u`
    in Haskell >>= oder do-Ausdrücke
- Huh?

Monads: Beispiel IO

IO-Funktionen

```
putStrLn :: String -> IO ()
getLine :: IO String
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Monads: Beispiel IO (2)

cat

```
let main = getLine >>= (
        \x -> putStrLn x >>= (
            \x -> main))
```

Monads: Beispiel IO (2)

cat

```
let main = getLine >>= (
        \x -> putStrLn x >>= (
            \x -> main))
```

cat in do-Notation

```
let main = do
        s <- getLine
        putStrLn s
        main
```

Module

Moduldefinition am Start der Datei

```
module Name(Name1, Name2, ...) where
```

Module

Moduldefinition am Start der Datei

```
module Name (Name1, Name2, ...) where
```

Moduleinbindung

```
import Name
import Name (Name1, Name2, ...)
import Name hiding (Name1, Name2, ...)
```

Module

Moduldefinition am Start der Datei

```
module Name (Name1, Name2, ...) where
```

Moduleinbindung

```
import Name
import Name (Name1, Name2, ...)
import Name hiding (Name1, Name2, ...)
```

Modulname korrespondiert mit Dateipfad

Datei: `MyProg/Network.hs`

```
module MyProg.Network where
```

Einbindung: `import MyProg.Network`

Fahrplan

Vielen Dank fürs Wachbleiben!

Noch Fragen?