Akademisch und doch nützlich:

Einführung in die funktionale Programmierung

Lukas Braun (koomi)
koomi@hackerspace-bamberg.de

Vorwort

- "Funktionale Programmierung" ist ein Paradigma
- "Imperativ" ist das vorherrschende Paradigma
- Die Welt ist nicht in funktional und nicht funktional eingeteilt, funktionale Ideen existieren in imperativen Sprachen und andersrum.
- Typsignaturen und Syntax:

```
>  5 :: Int
>  add :: Int -> Int -> Int
>  add x y = x + y
>  add 2 3
```

- theoretische Einführung

Eigenschaften funktionaler Programmiersprachen

- Funktionen wie in der Mathematik
- keine Zuweisung, kein Zustand
- Programme bestehen nur aus Funktionsdefinitionen, -zusammensetzung und -anwendung (und Daten)
- Funktionen sind Daten, können über- und zurückgegeben, zur Runtime erstellt werden

```
> map :: (a -> b) -> [a] -> [b]
> map plusOne [1,2,3]  =>  [2,3,4]
```

- Currying & partielle Anwendung
- algebraische Datentypen & Pattern Matching
- statt Schleifen häufig Rekursion
- IO/Seiteneffekte werden durch ausgefallene Tricks so in die Sprache integriert, dass die Reinheit bestehen bleibt (z.B. Monaden)

Erhaltene Vorteile

- mathematisch beschreibbar bis hin zu formalen Korrektheitsbeweisen
- Die Reinfolge der Berechnung ist irrelevant
  -> problemloser Parallelismus, da kein Zustand und folglich keine Race Conditions etc.
  -> Bedarfsauswertung (lazy evaluation)
- Kontrollstrukturen durch Higher-Order Functions und Lazyness leicht selbst implementierbar
- sehr mächtige Typsysteme mit Typinferenz möglich
- Der Fokus liegt mehr darauf, was rauskommen soll, nicht wie dies erreicht wird.

Funktionsbegriff in Mathematik vs. Informatik

- Informatik: Funktion als Unterprogramm,
  kann Variablen verändern, I/O, etc.
  -> Bei gleicher Eingabe zu unterschiedlichen Zeitpunkten sind unterschiedliche Ergebnisse möglich.
  -> Ein Funktionsaufruf kann andere Funktionen beeinflussen, auch wenn kein direkter Zusammenhang besteht.

- Mathe: Abbildung von Eingabewert auf Ausgabewert
  -> Keine "Nebeneffekte" (side effects)
  -> Bei gleicher Eingabe stets gleiches Ergebnis (referenzielle Transparenz)

Currying und partielle Anwendung

Currying:
= Funktionen mit mehreren Argumenten sind tatsächlich
  geschachtelte Funktionen mit einem Argument

```
> add :: Int ->  Int -> Int
> add :: Int -> (Int -> Int)
```

partielle Anwendung:
= Nicht alle Argumente einer mehrstelligen Funktion
  werden übergeben, um die erhaltene Funktion weiter
  zu verwenden.

```
> plusOne = add 1
> map (add 1) [1,2,3]
```

Algebraische Datentypen & Pattern Matching

algebraische Datentypen:
= Benutzerdefinierte Datentypen zusammengesetzt aus bestehenden Typen und Konstruktoren

```
> data Int = 0 | 1 | 2 | ...
> data List a = Cons a (List a) | Nil
> numbers :: List Int
> numbers = Cons 1 (Cons 2 Nil)
```

Pattern Matching:
= Dekonstruktion von ADT durch Matching gegen die Konstruktoren

```
> map :: (a -> b) -> List a -> List b
> map f Nil = Nil
> map f (Cons x xs) = Cons (f x) (map f xs)
```

Rekursion

- gefahrlos möglich durch Tail Call Optimization:
  Kein neuer Stack-Frame benötigt bei Funktions-
  aufruf als Rückgabe einer anderen Funktion
  -> Umwandlung in iterativen Algorithmus
- Ermöglichung von TCO durch Umwandlungen:

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

  wird zu:

```
> fac' 0 x = x
> fac' n x = fac (n-1) (x*n)
```

- Häufige Rekursions-Schemen werden in Higher-Order
  Functions ausgelagert (map, folds, filter, ...)

Lazy Evaluation

= Ausdrücke werden erst berechnet, wenn der Wert tatsächlich benötigt wird, vorher als "Thunks" (Versprechen) gespeichert

- unbenötigte Werte werden nicht berechnet
- unendliche Datenstrukturen
- Parameter bei Funktionsaufruf nicht berechnet
  -> Funktionen als Kontrollstrukturen
- erschwert das Nachdenken über Speicherverbrauch
- Space Leak: Ergebnisse der Thunks würden weniger Speicher benötigen, mögl. linearer Anstieg statt konstantem Speicherverbrauch
- unvereinbar mit imperativem Programmieren da Zeitpunkt bei Nebeneffekten entscheidend

Benutzerdefinierte Kontrollstrukturen

```
> if :: Bool -> a -> a -> a
> if True  exp exp' = exp
> if False exp exp' = exp'
> foo = if True (f a) (g b)
```

Bedarfsauswertung:
Der Bool und die Thunks von (f a) und (f b) werden an if übergeben, if gibt den Thunk (f a) zurück.

Strikte Auswertung:
Beim Aufruf von if werden (f a) und (f b) berechnet und deren Ergebnisse übergeben, das Resultat von (f a) wird zurückgegeben.

Typsysteme

- untypisiertes Lambda-Kalkül in Lisps
- seit Hindley-Milner-Typinferenz verstärkt starke statische Typsysteme, jedoch ohne Typen angeben zu müssen. Beispiel:
  Berechnungen die scheitern können werden durch Typen gekennzeichnet und Behandlung erzwungen

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

- Dependent Types: Typen können von Werten abhängen
- Curry-Howard-Korrespondenz: Typen sind logische Aussagen, ein Programm mit diesem Typ ist ein Beweis für diese Aussage. Code-Inferenz ist möglich.

Monadisches IO

- Funktionen die IO ausführen geben ein IO a zurück

```
> readFile :: String -> IO String
```

- Pattern Matching gegen IO geht nicht, Werte können nur durch monadische Funktionen benutzt werden:

```
> bind :: IO a -> (a -> IO b) -> IO b
> bind = >>=
> return :: a -> IO a
> >> :: IO a -> IO b -> IO b
> putStrLn "Schreib was" >> getLine >>= putStrLn
```

- Syntaktischer Zucker:

```
> main = do
>   putStrLn "Schreib was"
>   input <- getLine
>   putStrLn input
```

Links

- Einführung in Haskell mit viel Code-Beispielen: http://goo.gl/RbmM7
- Andreas Bogk programmiert einen zertifizierten PDF-Parser mit Coq: http://goo.gl/fFYdt