Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 12.01.2011: Aktionen und Zustände

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

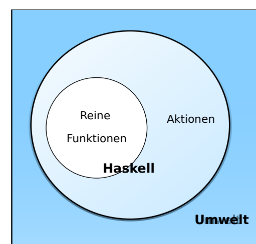
Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Aktionen und Zustände
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Ein/Ausgabe in funktionale Sprachen
- Wo ist das Problem?
- Aktionen und der Datentyp *IO*.
- Aktionen als Werte
- Aktionen als Zustandstransformationen

Ein- und Ausgabe in funktionalen Sprachen



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- readString :: ... → String ??

Lösung:

- Seiteneffekte am Typ erkennbar
- Aktionen können nur mit Aktionen komponiert werden
- „einmal Aktion, immer Aktion"

Aktionen als abstrakter Datentyp

- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO α

(>>=)   :: IO α → (α→ IO β) → IO β

return  :: α→ IO α
```

- Plus elementare Operationen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von stdin lesen:

```
getLine   :: IO String
```

- Zeichenkette auf stdout ausgeben:

```
putStr    :: String→ IO ()
```

- Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String→ IO ()
```


Einfache Beispiele

- Echo einfach

```
echo1 :: IO ()
echo1 = getLine >>= putStrLn
```

- Echo mehrfach

```
echo :: IO ()
echo = getLine >>= putStrLn >>= λ_ → echo
```

- Was passiert hier?
  - Verknüpfen von Aktionen mit >>=
  - Jede Aktion gibt Wert zurück

Noch ein Beispiel

- Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine
       >>= λs→ putStrLn (reverse s)
       >> ohce
```

- Was passiert hier?
  - Reine Funktion reverse wird innerhalb von Aktion putStrLn genutzt
  - Folgeaktion ohce benötigt Wert der vorherigen Aktion nicht
  - Abkürzung: >>

```
p >> q = p >>= λ_ → q
```

Die **do**-Notation

- Syntaktischer Zucker für IO:

```
echo =                          echo =
  getLine                  ⟺      do s← getLine
  >>= λs→ putStrLn s                  putStrLn s
  >> echo                             echo
```

  - Rechts sind >>=, >> implizit.
- Es gilt die Abseitsregel.
  - Einrückung der ersten Anweisung nach **do** bestimmt Abseits.

Drittes Beispiel

- Zählendes, endliches Echo

```
echo3 :: Int→ IO ()
echo3 cnt = do
  putStr (show cnt ++ ": ")
  s← getLine
  if s /= "" then do
      putStrLn $ show cnt ++ ": "++ s
      echo3 (cnt+ 1)
    else return ()
```

- Was passiert hier?
  - Kombination aus Kontrollstrukturen und Aktionen
  - Aktionen als Werte
  - Geschachtelte **do**-Notation

Module in der Standardbücherei

- Ein/Ausgabe, Fehlerbehandlung (Modul IO)
- Zufallszahlen (Modul Random)
- Kommandozeile, Umgebungsvariablen (Modul System)
- Zugriff auf das Dateisystem (Modul Directory)
- Zeit (Modul Time)

Ein/Ausgabe mit Dateien

- Im Prelude vordefiniert:
  - Dateien schreiben (überschreiben, anhängen):

```
type FilePath = String
writeFile     :: FilePath → String → IO ()
appendFile    :: FilePath → String → IO ()
```

- Datei lesen (verzögert):

```
readFile      :: FilePath → IO String
```

- Mehr Operationen im Modul IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String→ IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file ++ ": "++
                show (length (lines cont),
                      length (words cont),
                      length cont)
```

- Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: wc verbessert.

- Effizienter: Dateiinhalt einmal traversieren

```
cnt :: Int→ Int→ Int→ Bool→ String
       → (Int, Int, Int)
cnt l w c _ [] = (l, w, c)
cnt l w c bl (x:xs)
  | isSpace x && not bl = cnt l' (w+1) (c+1) True xs
  | isSpace x && bl     = cnt l' w (c+1) True xs
  | otherwise           = cnt l w (c+1) False xs where
    l' = if x == '\n' then l+1 else l
```

- Hauptprogramm:

```
wc :: String→ IO ()
wc file = do
 cont ← readFile file
 putStrLn $ file ++ ": "++ show (cnt 0 0 0 False cont)
```

- Datei wird verzögert gelesen und dabei verbraucht.

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich.
- Endlosschleife:

```
forever :: IO α→ IO α
forever a = a >> forever a
```

- Iteration (feste Anzahl):

```
forN :: Int→ IO α→ IO ()
forN n a | n == 0    = return ()
         | otherwise = a >> forN (n−1) a
```

- Vordefinierte Kontrollstrukturen (Control.Monad):
  - when, mapM, forM, sequence, ...

Fehlerbehandlung

- Fehler werden durch IOError repräsentiert
- Fehlerbehandlung durch Ausnahmen (ähnlich Java)

```
ioError :: IOError → IO α      -- "throw"
catch   :: IO α→ (IOError→ IO α) → IO α
```

- Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- Fehlerbehandlung für wc:

```
wc2 :: String→ IO ()
wc2 file =
  catch (wc file)
        (λe→ putStrLn $ "Fehler:␣"++ show e)
```

- IOError kann analysiert werden (siehe Modul IO)
- read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a⇒ String→ IO a
```


So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α)→ IO α
```

  - Warum ist randomIO Aktion?
- Beispiel: Aktionen zufällig oft ausführen

```
atmost :: Int→ IO α→ IO [α]
atmost most a =
  do l← randomRIO (1, most)
     mapM id (replicate l a)
```

- Zufälligen String erzeugen

```
randomStr :: IO String
randomStr = atmost 40 (randomRIO ('a','z'))
```


Ausführbare Programme

- Eigenständiges Programm ist Aktionen
- Hauptaktion: main in Modul Main
- wc als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Char(isSpace)

main = do
  args ← getArgs
  mapM wc2 args
```


Funktionen mit Zustand

Theorem (Currying)

Folgende Typen sind isomorph:

$$A \times B \to C \cong A \to B \to C$$

- In Haskell: folgende Funktionen sind invers:

```
curry   :: ((α, β) → γ)→ α→ β→ γ
uncurry :: (α→ β→ γ)→ (α, β) → γ
```


Funktionen mit Zustand

- Idee: Seiteneffekt explizit machen
- Funktion $f : A \to B$ mit Seiteneffekt in Zustand S:

$$f : A \times S \to B \times S$$
$$\cong$$
$$f : A \to S \to B \times S$$

- Datentyp: $S \to B \times S$
- Komposition: Funktionskomposition und uncurry

In Haskell: Zustände explizit

- Datentyp: Berechnung mit Seiteneffekt in Typ Σ:

```
type State Σ α = Σ→ (α, Σ)
```

- Komposition zweier solcher Berechnungen:

```
comp :: State Σ α→ (α→ State Σ β)→ State Σ β
comp f g = uncurry g ∘ f
```

- Lifting:

```
lift :: α→ State Σ α
lift = curry id
```


Beispiel: Ein Zähler

- Datentyp:

```
type WithCounter α = State Int α
```

- Zähler erhöhen:

```
tick :: WithCounter ()
tick i = ((), i+1)
```

- Zähler auslesen:

```
read :: WithCounter Int
read i = (i, i)
```

- Zähler zurücksetzen:

```
reset :: WithCounter ()
reset i = ((), 0)
```


Implizite vs. explizite Zustände

- Nachteil: Zustand ist explizit
  - Kann dupliziert werden
- Daher: Zustand implizit machen
  - Datentyp verkapseln
  - Signatur State, comp, lift, elementare Operationen

Aktionen als Zustandstransformationen

- Idee: Aktionen sind Transformationen auf Systemzustand S
- S beinhaltet
  - Speicher als Abbildung $A \rightharpoonup V$ (Adressen A, Werte V)
  - Zustand des Dateisystems
  - Zustand des Zufallsgenerators
- In Haskell: Typ RealWorld
  - "Virtueller" Typ, Zugriff nur über elementare Operationen
  - Entscheidend nur Reihenfolge der Aktionen

Zusammenfassung

- Ein/Ausgabe in Haskell durch Aktionen
- Aktionen (Typ IO α) sind seiteneffektbehaftete Funktionen
- Komposition von Aktionen durch

```
(>>=)   :: IO α → (α → IO β) → IO β
return  :: α → IO α
```

- **do**-Notation
- Fehlerbehandlung durch Ausnahmen (IOError, catch).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: getLine, putStr, putStrLn, readFile, writeFile
  - Module: IO, Random
- Aktionen sind implementiert als Zustandstransformationen