

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

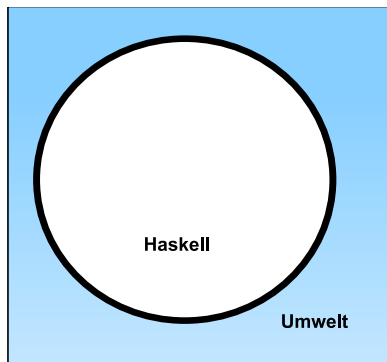
Fahrplan

- ▶ 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**

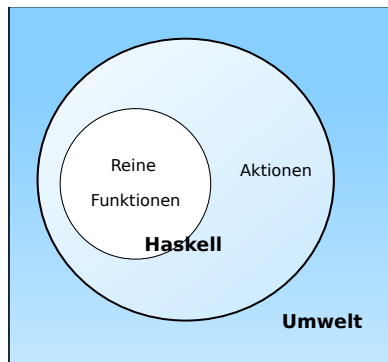
Ein- und Ausgabe in funktionalen Sprachen



Problem:

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

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  $\alpha$ 
```

```
( $\gg=$ )      :: IO  $\alpha \rightarrow (\alpha \rightarrow \text{IO } \beta) \rightarrow \text{IO } \beta$ 
```

```
return    ::  $\alpha \rightarrow \text{IO } \alpha$ 
```

- ▶ 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 >>= \_ → echo
```

▶ Was passiert hier?

- ▶ **Verknüpfen** von Aktionen mit $\gg=$
- ▶ Jede Aktion gibt **Wert** zurück

Noch ein Beispiel

- ▶ Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine
      >>= \s → 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 >>= \_ → q
```

Die **do**-Notation

- ▶ Syntaktischer Zucker für IO:

```
echo =  
  getLine  
  >>= λs → putStrLn s  
  >> echo
```



```
echo =  
  do s ← getLine  
      putStrLn s  
      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 Standardbibliothek

- ▶ `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
```

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)
    (\e → 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**?

So ein Zufall!

- ▶ Zufallswerte:

```
randomRIO :: ( $\alpha$ ,  $\alpha$ )  $\rightarrow$  IO  $\alpha$ 
```

- ▶ Warum ist randomIO **Aktion**?

- ▶ **Beispiel**: Aktionen zufällig oft ausführen

```
atmost :: Int  $\rightarrow$  IO  $\alpha$   $\rightarrow$  IO [ $\alpha$ ]  
atmost most a =  
  do l  $\leftarrow$  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*:

$$A \times B \rightarrow C \cong A \rightarrow B \rightarrow C$$

► In Haskell: folgende Funktionen sind *invers*:

```
curry    :: (( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ )  $\rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$   $\gamma$   
uncurry  :: ( $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$   $\gamma$ )  $\rightarrow$  ( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ 
```

Funktionen mit Zustand

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

$$\begin{aligned} f : A \times S &\rightarrow B \times S \\ &\cong \\ f : A &\rightarrow S \rightarrow B \times S \end{aligned}$$

- ▶ Datentyp: $S \rightarrow B \times S$
- ▶ Komposition: Funktionskomposition und **uncurry**

In Haskell: Zustände **explizit**

- ▶ Datentyp: Berechnung mit Seiteneffekt in Typ Σ :

```
type State  $\Sigma$   $\alpha = \Sigma \rightarrow (\alpha, \Sigma)$ 
```

- ▶ Komposition zweier solcher Berechnungen:

```
comp :: State  $\Sigma$   $\alpha \rightarrow (\alpha \rightarrow \text{State } \Sigma \beta) \rightarrow \text{State } \Sigma \beta$   
comp f g = uncurry g  $\circ$  f
```

- ▶ Lifting:

```
lift ::  $\alpha \rightarrow \text{State } \Sigma \alpha$   
lift = curry id
```

Beispiel: Ein Zähler

- ▶ Datentyp:

```
type WithCounter  $\alpha$  = State Int  $\alpha$ 
```

- ▶ 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 $A \rightarrow V$ (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 $\text{IO } \alpha$) sind seiteneffektbehaftete Funktionen
- ▶ **Komposition** von Aktionen durch

```
(\gg=)    :: IO \alpha \to (\alpha \to IO \beta) \to IO \beta  
return   :: \alpha \to IO \alpha
```

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