

# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 10 vom 18.12.2018: Aktionen und Zustände

Christoph Lüth

Universität Bremen

Wintersemester 2018/19



## Organisatorisches

- ▶ Probeklausur
  - ▶ Echte Klausur: 2 Stunden, wahrscheinlich 4 Programmieraufgaben
  - ▶ Probeklausur und Fragen werden **veröffentlicht**.
- ▶ Tutorium Do 16– 18: **Raumänderung** (diese Woche in MZH 1450)



## Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
  - ▶ **Aktionen und Zustände**
  - ▶ Monaden als Berechnungsmuster
  - ▶ Domänenspezifische Sprachen (DSLs)
  - ▶ Scala — Eine praktische Einführung
  - ▶ Rückblick & Ausblick

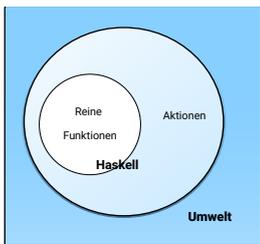


## Inhalt

- ▶ Ein/Ausgabe in funktionale Sprachen
- ▶ Wo ist das **Problem**?
- ▶ **Aktionen** und der Datentyp *IO*.
- ▶ Vordefinierte Aktionen
- ▶ Beispiel: Wortratespiel
- ▶ Aktionen als Werte



## 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
```
- ▶ Dazu **elementare** Aktionen (lesen, schreiben etc)



## Elementare Aktionen

- ▶ Zeile von Standardeingabe (stdin) **lesen**:

```
getLine :: IO String
```

- ▶ Zeichenkette auf Standardausgabe (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
      >>= \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
  >>= \s → putStrLn s
  >> 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



## 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
```

- ▶ "Lazy I/O": Zugriff auf Dateien erfolgt **verzögert**

- ▶ Interaktion von nicht-striktiger Auswertung mit zustandsbasiertem Dateisystem kann überraschend sein



## Ein/Ausgabe mit Dateien: Abstraktionsebenen

- ▶ **Einfach**: readFile, writeFile

- ▶ **Fortgeschritten**: Modul System.IO der Standardbibliothek

- ▶ Buffered/Unbuffered, Seeking, &c.
- ▶ Operationen auf Handle

- ▶ **Systemnah**: Modul System.Posix

- ▶ Filedeskriptoren, Permissions, special devices, etc.



## 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)
```

- ▶ Datei wird gelesen
- ▶ Anzahl Zeichen, Worte, Zeilen gezählt
- ▶ Erstaunlich (hinreichend) effizient



## 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
```



## Kontrollstrukturen

- ▶ Vordefinierte Kontrollstrukturen (Control.Monad):

```
when :: Bool → IO () → IO ()
```

- ▶ Sequenzierung:

```
sequence :: [IO α] → IO [α]
```

- ▶ Sonderfall: [()] als ()

```
sequence_ :: [IO ()] → IO ()
```

- ▶ Map und Filter für Aktionen:

```
mapM :: (α → IO β) → [α] → IO [β]
mapM_ :: (α → IO ()) → [α] → IO ()
filterM :: (α → IO Bool) → [α] → IO [α]
```



## Fehlerbehandlung

- ▶ Fehler werden durch Exception repräsentiert (Modul `Control.Exception`)
  - ▶ Exception ist **Typklasse** — kann durch eigene Instanzen erweitert werden
  - ▶ Vordefinierte Instanzen: u.a. `IOError`
- ▶ Fehlerbehandlung durch **Ausnahmen** (ähnlich Java)

```
throw :: Exception  $\gamma \rightarrow \gamma \rightarrow \alpha$ 
catch :: Exception  $\gamma \rightarrow IO \alpha \rightarrow (\gamma \rightarrow IO \alpha) \rightarrow IO \alpha$ 
try :: Exception  $\gamma \rightarrow IO \alpha \rightarrow IO (Either \gamma \alpha)$ 
```
- ▶ Faustregel: `catch` für unerwartete Ausnahmen, `try` für erwartete
- ▶ Ausnahmen überall, Fehlerbehandlung **nur in Aktionen**

PI3 WS 18/19

17 [25]



## Fehler fangen und behandeln

"Ask forgiveness not permission"

Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als Fehlerbedingungen abzufragen

- ▶ Fehlerbehandlung für `wc`:

```
wc2 :: String  $\rightarrow IO ()$ 
wc2 file =
  catch (wc file)
    (\e  $\rightarrow$  putStrLn $ "Fehler:␣" ++ show (e :: IOError))
```

- ▶ `IOError` kann analysiert werden (siehe `System.IO.Error`)
- ▶ `read` mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read  $\alpha \rightarrow String \rightarrow IO \alpha$ 
```

PI3 WS 18/19

18 [25]



## Ausführbare Programme

- ▶ Eigenständiges Programm ist **Aktion**
- ▶ **Hauptaktion**: `main :: IO ()` in Modul `Main`
  - ▶ ... oder mit der Option `-main-is M.f` setzen
- ▶ `wc` als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Control.Exception

...

main :: IO ()
main = do
  args  $\leftarrow$  getArgs
  mapM_ wc2 args
```

PI3 WS 18/19

19 [25]



## Beispiel: Traversal eines Verzeichnisbaums

- ▶ Verzeichnisbaum traversieren, und für jede Datei eine **Aktion** ausführen:

```
travFS :: (FilePath  $\rightarrow IO ()$ )  $\rightarrow$  FilePath  $\rightarrow IO ()$ 
travFS action p = catch (do
  cs  $\leftarrow$  getDirectoryContents p
  let cp = map (p </>) (cs \ \ [".", ".."])
      dirs  $\leftarrow$  filterM doesDirectoryExist cp
      files  $\leftarrow$  filterM doesFileExist cp
      mapM_ action files
      mapM_ (travFS action) dirs)
  (\e  $\rightarrow$  putStrLn $ "ERROR:␣" ++ show (e :: IOError))
```

- ▶ Nutzt Funktionalität aus `System.Directory`, `System.FilePath`

PI3 WS 18/19

20 [25]



## So ein Zufall!

- ▶ Zufallswerte:

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

- ▶ Warum ist `randomIO` **Aktion**?

- ▶ **Beispiele**:

- ▶ Aktion zufällig oft ausführen:

```
atmost :: Int  $\rightarrow IO \alpha \rightarrow IO [a]$ 
atmost most a =
  do l  $\leftarrow$  randomRIO (1, most)
     sequence (replicate l a)
```

- ▶ Zufälligen String erzeugen:

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

PI3 WS 18/19

21 [25]



## Fallbeispiel: Wörter raten

- ▶ Unterhaltungsprogramm: der Benutzer rät Wörter
- ▶ Benutzer kann einzelne Buchstaben eingeben oder das ganze Wort
- ▶ Wort wird maskiert ausgegeben, nur geratene Buchstaben angezeigt

PI3 WS 18/19

22 [25]



## Wörter raten: Programmstruktur

- ▶ Hauptschleife:

```
play :: String  $\rightarrow String \rightarrow String \rightarrow IO ()$ 
```

- ▶ Argumente: Geheimnis, geratene Buchstaben (enthalten, nicht enthalten)

- ▶ Benutzereingabe:

```
getGuess :: String  $\rightarrow String \rightarrow IO Char$ 
```

- ▶ Argumente: geratene Zeichen (im Geheimnis enthalten, nicht enthalten)

- ▶ Hauptfunktion:

```
main :: IO ()
```

- ▶ Liest ein Lexikon, wählt Geheimnis aus, ruft Hauptschleife auf

PI3 WS 18/19

23 [25]



## Zusammenfassung

- ▶ Ein/Ausgabe in Haskell durch **Aktionen**
- ▶ **Aktionen** (`Typ IO  $\alpha$` ) sind seiteneffektbehaftete Funktionen
- ▶ **Komposition** von Aktionen durch

```
( $\gg$ ) :: IO  $\alpha \rightarrow (\alpha \rightarrow IO \beta) \rightarrow IO \beta$ 
return ::  $\alpha \rightarrow IO \alpha$ 
```
- ▶ **do**-Notation
- ▶ Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`, `try`).
- ▶ Verschiedene Funktionen der Standardbücherei:
  - ▶ Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
  - ▶ Module: `System.IO`, `System.Random`
- ▶ Aktionen sind **implementiert** als **Zustandstransformationen**

PI3 WS 18/19

24 [25]



