Praktische Informatik 3: Funktionale Programmierung
Vorlesung 10 vom 20.12.2016: Aktionen und Zustände

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

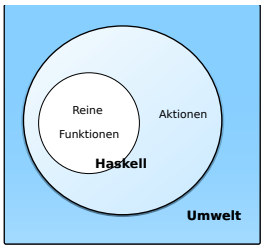
Fahrplan

- 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
  - Rückblich & 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
       >>= 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 =
  getLine
  >>= λs→ putStrLn s
  >> echo
```
$\Longleftrightarrow$
```
echo =
  do s← getLine
     putStrLn s
     echo
```

  - Rechts sind $\gg=$, $\gg$ 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 System.IO, Control.Exception)
- Zufallszahlen (Modul System.Random)
- Kommandozeile, Umgebungsvariablen (Modul System.Environment)
- Zugriff auf das Dateisystem (Modul System.Directory)
- Zeit (Modul System.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 System.IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle
- Noch mehr Operationen in 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)

```
catch  :: Exception γ⇒ IO α → (γ→ IO α) → IO α
try    :: Exception γ⇒ IO α → IO (Either γ α)
```

- Faustregel: catch für unerwartete Ausnahmen, try für erwartete
- 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))
```

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

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


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 ← getArgs
  mapM_ wc2 args
```


Beispiel: Traversion eines Verzeichnisbaums

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

```
travFS :: (FilePath→ IO ())→ FilePath→ IO ()
```

- Nutzt Funktionalität aus System.Directory, System.FilePath

```
travFS action p = do
  res ← try (getDirectoryContents p)
  case res of
    Left  e → putStrLn $ "ERROR: "++ show (e :: IOError)
    Right cs → do let cp = map (p </>) (cs \\ [".", ".."])
                  dirs  ← filterM doesDirectoryExist cp
                  files ← filterM doesFileExist cp
                  mapM_ action files
                  mapM_ (travFS action) dirs
```


So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α)→ IO α
```

  - Warum ist randomIO Aktion?
- Beispiele:
  - Aktion zufällig oft ausführen:

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

  - Zufälliges Element aus einer nicht-leeren Liste auswählen:

```
pickRandom :: [α]→ IO α
pickRandom [] = error "pickRandom: empty list"
pickRandom xs = do
  i ← randomRIO (0, length xs − 1)
  return $ xs !! i
```


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

Wörter raten: Programmstruktur

- Trennung zwischen Spiel-Logik und Nutzerschnittstelle
- Spiel-Logik (GuessGame):
  - Programmzustand:

```
data State = St { word   :: String — Zu ratendes Wort
                , hits   :: String — Schon geratene Buchstaben
                , miss   :: String — Falsch geratene Buchstaben
                }
```

  - Initialen Zustand (Wort auswählen):

```
initialState :: [String]→ IO State
```

  - Nächsten Zustand berechnen (Char ist Eingabe des Benutzers):

```
data Result = Miss | Hit | Repetition | GuessedIt | TooManyTries

processGuess :: Char→ State→ (Result, State)
```


Wörter raten: Nutzerschnittstelle

- Hauptschleife (play)
  - Zustand anzeigen
  - Benutzereingabe abwarten
  - Neuen Zustand berechnen
  - Rekursiver Aufruf mit neuem Zustand
- Programmanfang (main)
  - Lexikon lesen
  - Initialen Zustand berechnen
  - Hauptschleife aufrufen

```
play :: State→ IO ()
play st = do
  putStrLn (render st)
  c ← getGuess st
  case (processGuess c st) of
    (Hit, st)  → play st
    (Miss, st) → do putStrLn "Sorry, no."; play st
    (Repetition, st)→ do putStrLn "You already tried that."; play st
    (GuessedIt, st)→ putStrLn "Congratulations, you guessed it."
    (TooManyTries, st) →
      putStrLn $ "The word was "++ word st ++ " — you lose."
```


Kontrollumkehr

- Trennung von Logik (State, processGuess) und Nutzerinteraktion nützlich und sinnvoll
- Wird durch Haskell Tysystem unterstützt (keine UI ohne IO)
- Nützlich für andere UI mit Kontrollumkehr
- Beispiel: ein GUI für das Wörterratespiel (mit Gtk2hs)
  - GUI ruft Handler-Funktionen des Nutzerprogramms auf
  - Spielzustand in Referenz (IORef) speichern
- Vgl. MVC-Pattern (Model-View-Controller)

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, try).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: getLine, putStr, putStrLn, readFile, writeFile
  - Module: System.IO, System.Random
- Aktionen sind implementiert als Zustandstransformationen