Praktische Informatik 3: Funktionale Programmierung
Vorlesung 10 vom 18.01.2021: Aktionen und Zustände

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2020/21

Fahrplan

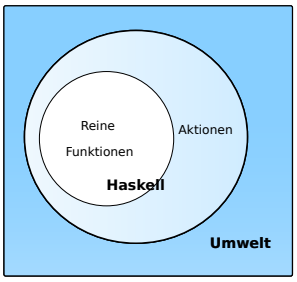
- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Funktionale Webanwendungen
  - Scala — Eine praktische Einführung
  - Rückblick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

Inhalt

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

I. Funktionale Ein/Ausgabe

Ein- und Ausgabe in funktionalen Sprachen



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- `readString` $::\ldots\to$ `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 $\gg=$
  - 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
```

⟺

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

- Rechts sind `>>=`, `>>` implizit
- Mit `←` gebundene Bezeichner **überlagern** vorherige
- 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

Zeit für eine Pause

Übung 10.1: Say My Name!

Wie sieht ein Haskell-Program aus, das erst nach dem Namen des Gegenübers fragt, und dann mit `Hallo, Christoph!` (oder was eingegeben wurde) freundlich grüßt?

Lösung:

```
greeter :: IO ()
greeter = do
  putStr "What's␣your␣name,␣love?␣"
  s ← getLine
  putStrLn $ "Hullo,␣" ++ s ++ ".␣Pleased␣to␣meet␣you."
```

- `putStr` statt `putStrLn` erlaubt „Prompting“
- Argumente von `putStrLn` klammern (oder `$`)

II. Aktionen als Werte

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]
```


Jetzt ihr!

Übung 10.2: Eine „While-Schleife“ in Haskell

Schreibt einen Kombinator

```
while :: IO Bool → IO α→ IO ()
```

der solange das zweite Argument (den Rumpf) auswertet wie das erste Argument zu `True` auswertet.

Lösung:

- Erste Lösung:

```
while c b = do a← c; if a then b >> while c b else return ()
```

- Vorteil: ist **endrekursiv**.
- Wieso eigentlich `IO ()`?

III. Ein/Ausgabe

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-strikter Auswertung mit zustandsbasiertem Dateisystem kann überraschend sein

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

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

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

Ein/Ausgabe mit Dateien: Abstraktionsebenen

- **Einfach**: `readFile`, `writeFile`
- **Fortgeschritten**: Modul `System.IO` der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf `Handle`
- **Systemnah:** Modul `System.Posix`
  - Filedeskriptoren, Permissions, special devices, etc.

IV. Ausnahmen und Fehlerbehandlung

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 γ⇒ γ→ α
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
- Ausnahmen überall, Fehlerbehandlung **nur in Aktionen**

Fehler fangen und behandeln

"Ask forgiveness not permission" (Grace Hopper)

Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als vorherige Abfrage von Fehlerbedingungen.

- Warum? Umwelt nicht **sequentiell**.
- 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
  putStrLn $ "Command␣line␣arguments:␣" ++ show args
  mapM_ wc2 args
```

Beispiel: Traversion eines Verzeichnisbaums

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

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

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

Alles zählt.

Übung 10.3: Alles zählt

Kombiniert `Traverse` und `WC` zu einem Programm

```
ls :: FilePath→ IO ()
```

welches in einem gegeben Verzeichnis den Inhalt aller darin enthaltenen Dateien zählt.

Lösung: `wc2` (mit Fehlerbehandlung) wird einfach die Traversionsfunktion:

```
ls = travFS wc2
```

Das ist alles.

V. Anwendungsbeispiel

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)
```

  - Zufälligen String erzeugen:

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

- Hinweis: Funktionen aus `System.Random` zu importieren, muss ggf. installiert werden.

Fallbeispiel: Wörter raten

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

Wörter raten: Programmstruktur

- Hauptschleife:

```
play :: String→ String→ String→ IO ()
```

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

```
getGuess :: String→ String→ IO Char
```

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

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

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

Nunc est ludendum.

Übung 10.3: Linguistic Interlude

Ladet den Quellcode herunter, übersetzt das Spiel und ratet fünf Wörter.
Wer noch etwas tun möchte, kann das Spiel so erweitern, dass es nachdem das Wort erfolgreich geraten wurde, ein neues Wort rät, und insgesamt zählt, wieviele Worte schon (nicht) geraten wurden.

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`
- Nächste Vorlesung: Wie sind Aktionen eigentlich **implementiert**? Schwarze Magie?