Praktische Informatik 3: Funktionale Programmierung

Vorlesung 10 (20.12.2022): Aktionen und Zustände

Christoph Lüth

Wintersemester 2022/23

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

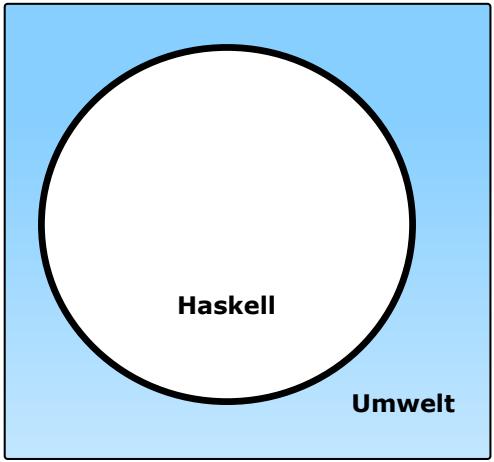
- Ein/Ausgabe in funktionale Sprachen
- Wo ist das **Problem**?
- **Aktionen** und der Datentyp *IO*.
- Vordefinierte Aktionen
- Beispiel: Wortratespiel
- Aktionen als Werte

Lernziele

Wir verstehen, wie wir Ein- und Ausgabe in Haskell funktional modellieren.

I. Funktionale Ein/Ausgabe

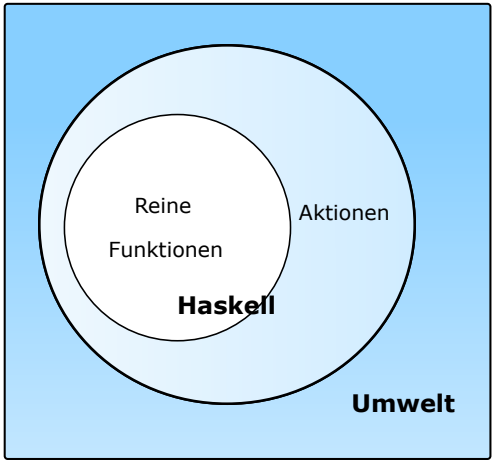
Ein- und Ausgabe in funktionalen Sprachen



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- `readString` $::$ `...` $\rightarrow$ `String` ??

Ein- und Ausgabe in funktionalen Sprachen



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- `readString` $::$ `...` $\rightarrow$ `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  -- Komposition

return :: α→ IO α                  -- Lifting
```

- 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
```

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**: $\gg$

```
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
- Mit $\leftarrow$ 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

☞ Siehe Übung 10.1

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]
```

☞ Siehe Übung 10.2

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`
  - Im `Prelude` **vordefiniert**, **portabel**.
- **Fortgeschritten**: Modul `System.IO` der Standardbücherei
  - Buffered/Unbuffered, Seeking, Operationen auf `Handle`
  - **Portabel** — funktioniert auf allen Platformen
- **Systemnah:** Modul `System.Posix`
  - Filedeskriptoren, Permissions, special devices, etc.
  - **Systemspezifisch** (nicht vollständig portabel).

IV. Ausnahmen und Fehlerbehandlung

Fehlerbehandlung, erster Versuch

- Wie könnten wir **Fehler** modellieren?

Fehlerbehandlung, erster Versuch

- Wie könnten wir **Fehler** modellieren?
  - Fehler werden durch Typ `E` repräsentiert.
  - Berechnung mit Fehler: `Either E` α
  - Fehler **fangen**: `catch :: Either E` $\alpha \rightarrow$ `(E` $\rightarrow \alpha) \rightarrow \alpha$
  - Fehler erzeugen: `Left e`

Fehlerbehandlung, erster Versuch

- Wie könnten wir **Fehler** modellieren?
  - Fehler werden durch Typ `E` repräsentiert.
  - Berechnung mit Fehler: `Either E` α
  - Fehler **fangen**: `catch :: Either E` $\alpha \rightarrow$ `(E`$\rightarrow \alpha) \rightarrow \alpha$
  - Fehler erzeugen: `Left e`
- Probleme:
  - Ausnahmen sollen **erweiterbar** bleiben.
  - Man muss **entweder alle** Berechnungen mit `Right x` in den Fehlertypen liften,
  - **oder** das Fangen ist nicht referentiell transparent.

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?

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`

☞ Siehe Übung 10.3

V. Anwendungsbeispiel

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

  - Warum ist `randomIO` **Aktion**?

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

☞ Siehe Übung 10.4

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?


Frohe Weihnachten und einen Guten Rutsch!