Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 07.01.09: Ein/Ausgabe in Funktionalen Sprachen

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
  - Abstrakte Datentypen
  - Signaturen & Axiome
  - Korrektheit von Programmen
  - Zustand und Aktionen
- Teil III: Beispiele, Anwendungen, Ausblicke

Inhalt

- Ein/Ausgabe in funktionale Sprachen
- Wo ist das Problem?
- Aktionen und der Datentyp *IO*.
- Beispiel: Worte zählen (`wc`)
- Aktionen als Werte

Ein- und Ausgabe in funktionalen Sprachen

Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- z. B. `readString :: ... -> String` ??

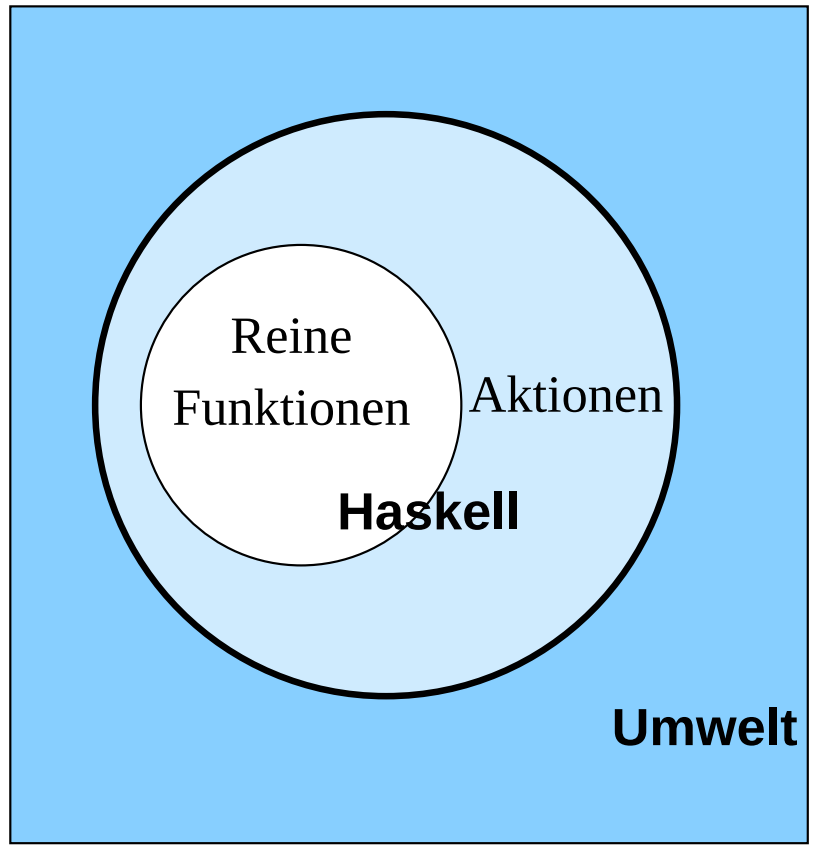
Ein- und Ausgabe in funktionalen Sprachen

Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- z. B. `readString :: ... -> String` ??

Lösung:

- Seiteneffekte am Typ erkennbar
- Aktionen können nur mit Aktionen komponiert werden
- „einmal Aktion, immer Aktion“



Modellierung des Systemzustands

- Idee: Aktionen sind Transformationen auf Systemzustand Σ
- Operationen:
  - Adressen A, Werte V
  - Lesen: `read` : $A \to \Sigma \to V \times \Sigma$
  - Schreiben: `write` : $A \to V \to \Sigma \to \Sigma$,
  - Äquivalent, aber besser `write` : $A \to V \to \Sigma \to () \times \Sigma$
- Eine Aktion ist $I \to \Sigma \to O \times \Sigma \quad \cong \quad I \times \Sigma \to O \times \Sigma$
  - Einheitliche Signatur für alle Aktionen
  - Parametrisiert über Ergebnistyp O

Komposition von Aktionen

- Komposition ist Funktionskomposition
- Wenn Rückgabewert der ersten Aktion p Eingabewert der zweiten q:

$$\frac{\begin{array}{l} p : A \to \Sigma \to B \times \Sigma \\ q : B \to \Sigma \to C \times \Sigma \end{array}}{\begin{array}{l} p\,;q : A \to \Sigma \to C \times \Sigma \\ (p\,;q)\ a\ S = \ \ let\ (b, S_1) = p\ a\ S \\ \qquad\qquad\quad\ \ in\ q\ b\ S_1 \end{array}}$$

- Lifting $\hat{a} : \Sigma \to A \times \Sigma$ für jedes $a : A$ definiert als $\hat{a}(S) = (a, S)$

Aktionen als abstrakter Datentyp

- Idee: `IO a` ist $\Sigma \longrightarrow a \times \Sigma$
- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO t

(>>=)  :: IO a -> (a-> IO b) -> IO b

return :: a-> IO a
```

- Plus elementare Operationen (lesen, schreiben etc)

Vordefinierte 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 >>=
  - 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
```

$\Longleftrightarrow$

```
echo =
  do s<- getLine
     putStrLn s
     echo
```

  - Rechts sind `>>=`, `>>` implizit.

- Es gilt die Abseitsregel.

  - Einrückung der ersten Anweisung nach `do` bestimmt Abseits.

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

Beispiel: Zeichen, Wörter, Zeilen zählen (`wc`)

```
wc :: String-> IO ()
wc file =
  do c <- readFile file
     putStrLn (show (length (lines c),
                     length (words c),
                     length c) ++
               " lines, words, characters.")
```

- Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: `wc` verbessert.

- Effizienter: Funktion `cnt`, die Dateiinhalt einmal traversiert

```
cnt :: Int-> Int-> Int-> Bool-> String-> (Int, Int, Int)
cnt l w c _ [] = (l, w, c)
cnt l w c blank (x:xs)
  | isSpace x && not blank = cnt l' (w+1) (c+1) True xs
  | isSpace x && blank     = 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.

- Mit `cnt` besseres `wc`:

```
wc' :: String-> IO ()
wc' file =
  do c <- readFile file
     putStrLn (show (cnt 0 0 0 False c)++
               " lines, words, characters.")
```

  - Datei wird verzögert gelesen und dabei verbraucht.

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich.
- Beispiele:
  - Endlosschleife:

```
forever :: IO a-> IO a
forever a = a >> forever a
```

  - Iteration (feste Anzahl):

```
forN :: Int-> IO a-> IO ()
forN n a | n == 0    = return ()
         | otherwise = a >> forN (n-1) a
```

Fehlerbehandlung

- Fehler werden durch `IOError` repräsentiert
- Fehlerbehandlung durch Ausnahmen (ähnlich Java)

```
ioError :: IOError -> IO a     -- "throw"
catch   :: IO a-> (IOError-> IO a) -> IO a
```

- Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- Fehlerbehandlung für `wc`:

```
wc2 :: String-> IO ()
wc2 file =
  catch (wc' file)
        (\e-> putStrLn ("Fehler beim Lesen: "++ show e))
```

- `IOError` kann analysiert werden (siehe Modul `IO`)
- `read` mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a=> String-> IO a
```

Map und Filter für Aktionen

- Map für Aktionen:

```
mapM :: (a-> IO b)-> [a]-> IO [b]
```

```
mapM_ :: (a-> IO ())-> [a]-> IO ()
```

- Filter für Aktionen
  - Importieren mit `import Monad (filterM)`.

```
filterM :: (a -> IO Bool) -> [a] -> IO [a]
```

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (a, a)-> IO a
```

  - Warum ist `randomIO` Aktion?

- Beispiel: Aktionen zufällig oft ausführen

```
atmost :: Int-> IO a-> IO [a]
atmost most a =
  do l<- 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)

main = do
  args <- getArgs
  mapM wc2 args
```

Zusammenfassung

- Ein/Ausgabe in Haskell durch Aktionen
- Aktionen (Typ `IO a`) sind seiteneffektbehaftete Funktionen
- Komposition von Aktionen durch

```
(>>=)  :: IO a-> (a-> IO b)-> IO b
return :: a-> IO a
```

- `do`-Notation
- Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
  - Module: `IO`, `Random`