Vorlesung vom 03.01.2005: Ein/Ausgabe in Funktionalen Sprachen

Inhalt

- Wo ist das Problem?
- Aktionen und der Datentyp IO.
- Vordefinierte Aktionen
- Beispiel: `Nim`
- Aktionen als Werte

Ein- und Ausgabe in funktionalen Sprachen

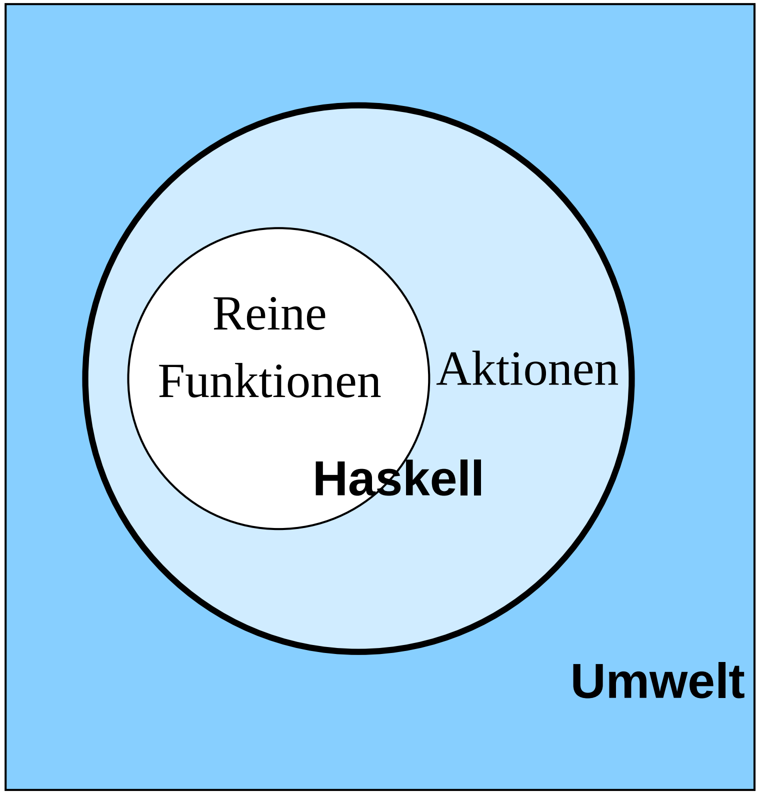
Problem: Funktionen mit Seiteneffekten nicht referentiell transparent.

- z. B. `readString :: ... -> String` ??

Lösung: Seiteneffekte am Typ `IO` erkennbar — Aktionen

- Aktionen können nur mit Aktionen komponiert werden
- „einmal IO, immer IO“

Aktionen als abstrakter Datentyp




```
type IO t

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

return :: a-> IO a
```

Vordefinierte Aktionen (Prelude)

- 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
```

- Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine >>= putStrLn . reverse >> ohce
```

Die do-Notation

- Vordefinierte Abkürzung:

```
(>>) :: IO t-> IO u-> IO u
f >> g = f >>= \_ -> g
```

- Syntaktischer Zucker für IO:

```
echo =                              echo =
  getLine                             do s<- getLine
  >>= \s-> putStrLn s       ⟺            putStrLn s
  >> echo                                 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, Worte, 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.")
```

- Testen.
- Nicht sehr effizient — Datei wird im Speicher gehalten.

Beispiel: `wc` verbessert.

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

- Datei wird verzögert gelesen und dabei verbraucht.

Ein längeres Beispiel: `Nim` revisited

- Benutzerschnittstelle von `Nim`:
- Am Anfang Anzahl der Hölzchen auswürfeln.
- Eingabe des Spielers einlesen.
- Wenn nicht zu gewinnen, aufgeben, ansonsten ziehen.
- Wenn ein Hölzchen übrig ist, hat Spieler verloren.

Alles Zufall?

- Zufallswerte: Modul `Random`, Klasse `Random`

```
class Random a where
  randomRIO :: (a, a)-> IO a
  randomIO  :: IO a
```

- Warum ist `randomIO` Aktion?
  - Referentielle Transparenz erlaubt keinen Nichtdeterminismus.
- Instanzen von `Random`: Basisdatentypen.
- `Random` enthält ferner
  - Zufallsgeneratoren für Pseudozufallszahlen.
  - Unendliche Listen von Zufallszahlen.

`Nim` **revisited**

- Importe und Hilfsfunktionen:

```
import Random (randomRIO)
```

- `wins` liefert `Just` n, wenn Zug n gewinnt; ansonsten `Nothing`

```
wins :: Int-> Maybe Int
wins n =
  if m == 0 then Nothing else Just m where
     m = (n- 1) `mod` 4
```

Hauptfunktion

- Start des Spiels mit n Hölzchen

```
play :: Int-> IO ()
play n =
  do putStrLn ("Der Haufen enthält "++ show n ++
               " Hölzchen.")
     if n== 1 then putStrLn "Ich habe gewonnen!"
       else
         do m<- getInput
            case wins (n-m) of
              Nothing -> putStrLn "Ich gebe auf."
              Just l  -> do putStrLn ("Ich nehme "
                                      ++ show l)
                            play (n-(m+l))
```

Benutzereingabe

- Zu implementieren: Benutzereingabe

```
getInput' :: IO Int
getInput' =
  do putStr "Wieviele nehmen Sie? "
     n <- do s<- getLine
             return (read s)
     if n<= 0 || n>3 then
       do putStrLn "Ungültige Eingabe!"
          getInput'
       else return n
```

- Nicht sehr befriedigend: Abbruch bei falscher Eingabe.

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

- Fangbare Benutzerfehler mit

```
userError::String-> IOError
```

- `IOError` kann analysiert werden— Auszug aus Modul `IO`:

```
isDoesNotExistError  :: IOError -> Bool
isIllegalOperation   :: IOError -> Bool
isPermissionError    :: IOError -> Bool
isUserError          :: IOError -> Bool
ioeGetErrorString    :: IOError -> String
ioeGetFileName       :: IOError -> Maybe FilePath
```

- `read` mit Ausnahme bei Fehler (statt Programmabbruch):

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

Robuste Eingabe

```
getInput :: IO Int
getInput =
  do putStr "Wieviele nehmen Sie? "
     n <- catch (do s<- getLine
                    readIO s)
                (\_ -> do putStrLn "Eingabefehler!"
                          getInput)
     if n<= 0 || n>3 then
       do putStrLn "Ungültige Eingabe!"
          getInput
       else return n
```

Haupt- und Startfunktion

- Begrüßung,
- Anzahl Hölzchen auswürfeln,
- Spiel starten.

```
main :: IO ()
main = do putStrLn "\nWillkommen bei Nim!\n"
          n <- randomRIO(5,49)
          play n
```

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich.
- Besser als jede imperative Sprache.

Beispiel: Kontrollstrukturen

- Endlosschleife:

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

- Iteration (variabel, wie `for` in Java)

```
for :: (a, a-> Bool, a-> a)-> (a-> IO ())-> IO ()
for (start, cont, next) cmd =
  iter start where
    iter s = if cont s then cmd s >> iter (next s)
                else return ()
```

Vordefinierte Kontrollstrukturen (Prelude)

- Listen von Aktionen sequenzieren:

```
sequence :: [IO a]-> IO [a]
sequence (c:cs) = do x  <- c
                     xs <- sequence cs
                     return (x:xs)
```

- Sonderfall: `[()]` als `()`

```
sequence_ :: [IO ()]-> IO ()
```

Map und Filter für Aktionen

- Map für Aktionen:

```
mapM :: (a-> IO b)-> [a]-> IO [b]
mapM f = sequence . map f

mapM_ :: (a-> IO ())-> [a]-> IO ()
mapM_ f = sequence_ . map f
```

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

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

Beispiel

- Führt Aktionen zufällig oft aus:

```
atmost :: Int-> IO a-> IO [a]
atmost most a =
  do l<- randomRIO (1, most)
     sequence (replicate l a)
```

- Zufälligen String:

```
randomStr :: IO String
randomStr = atmost 10 (randomRIO ('a','z'))
```

Zeigen.

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`
  - Module: `IO`, `Random`,