Praktische Informatik 3: Funktionale Programmierung

Vorlesung 9 / 19. Dezember 2023

I/O, Aktionen und Zustände

Thomas Barkowsky

Wintersemester 2023/24

Nachlese Probeklausur

- Ergebnisse veröffentlicht
- Durchschnitt: 1,7 von 10 Punkten
- Bestehensquote: 6,1 %

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- I/O, Aktionen und Zustände
- Testen und Qualitätssicherung
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- I/O in funktionalen Sprachen: Wo ist das Problem?
- Vordefinierte I/O Aktionen...
  - und ihre Verwendung in der richtigen Reihenfolge
- Umgang mit Dateien
- Zufallszahlen in Haskell
- Fallbeispiel

I/O in funktionalen Sprachen

I/O ist ein Problem! – Warum?

- Bisher: referentielle Transparenz!
  - Ausgabewerte von Funktionen hängen ausschließlich von den aktuellen Eingabewerten ab, nicht vom Kontext des Aufrufs
  - Dieselben Eingabewerte erzeugen immer dieselbe Ausgabe
  - Keine Seiteneffekte, keine versteckten Zustände
  - Alle Abhängigkeiten explizit in Funktionsdefinition
- Davon müssen wir uns jetzt verabschieden, denn...
  - Nutzereingaben sind beliebig
  - Der Inhalt von Dateien kann sich ändern
  - Ausgaben des Programms verändern den Systemzustand (Seiteneffekte!)
  - etc.

I/O ist ein Problem! – Warum noch?

- Zustandsänderungen des Gesamtsystems müssen in bestimmter **Reihenfolge** vorgenommen werden
  - z.B.: Programmreaktion aufgrund einer Nutzeraktion
  - "durchreichen" von Zustandsänderungen erforderlich
  - *Lazy evaluation* funktioniert nicht mehr

- Wie kriegen wir das in Haskell hin?

I/O in Haskell

- Strikte Trennung von **reinen** Funktionen (*pure functions*) und I/O
- Funktionen mit Seiteneffekten werden in Haskell **Aktionen** genannt
- Aktionen werden in einem gesonderten abstrakten Datentyp `IO a` gekapselt: `data IO a = ...  -- abstract` (aus *standard prelude*)
  - Seiteneffekte sind also am Datentyp erkennbar!
  - Aktionen können nur mit anderen Aktionen kombiniert werden
  - Abstrakt: Konstruktoren nicht sichtbar für Nutzer
- Beispiele:
  - `... :: IO String` = Aktion, die einen String zurückliefert
  - `... :: IO ()` = Aktion ohne Rückgabewert (ausschließlich Seiteneffekt)

Elementare Aktionen

Elementare I/O Aktionen

- Zeichen aus Tastatur lesen
- analog ganze Zeile
- Zeichen auf Bildschirm ausgeben
- analog ganze Zeile
- dito mit Zeilenvorschub

```
getChar  :: IO Char
getLine  :: IO String
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

- Umwandlung Wert in Aktion

```
return   :: a -> IO a
```

- Rückumwandlung aus Aktion? – NEIN! (einmal IO – immer IO)

Kombination von I/O Aktionen

- Auf die Reihenfolge kommt es an! (siehe oben)
- *bind* - Operator: `(>>=) :: IO a -> (a -> IO b) -> IO b`
  - Komposition von Aktionen:
    Für zwei Aktionen `c` und `d` führt `c >>= d` zuerst Aktion `c` aus;
    das Ergebnis wird an `d` übergeben
  - ähnlich wie Funktionskomposition
  - z.B. `getLine >>= putStrLn`

- ähnlich: `(>>) :: IO a -> IO b -> IO b`
  - *bind* für nachgelagerte Funktionen, die kein Ergebnis übernehmen
  - z.B. `getLine >>= putStr >> putStr " ++stop++\n"`

Aktionen und (reine) Funktionen

- Aktionen werden verwendet wie Funktionen, z.B. Rekursion:

```
echoLn :: IO ()
echoLn = getLine >>= putStrLn

echoLnR :: IO ()
echoLnR = echoLn >> echoLnR
```

- Kombination mit reinen Funktionen:

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

do - Notation

- Vereinfachende Schreibweise für Aktionen
- Fast wie imperative Programmierung (*syntactic sugar*):

```
echoLnR = getLine >>=
          putStrLn >>
          echoLnR
```

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

  - Schlüsselwort `do`
  - v_i `<-` a_i : **Generatoren**
  - v_i `<-` entfällt, wenn Ergebnis nicht benötigt wird
  - Alignierung der Anweisungen (Abseitsregel)

`do` – Notation: Beispiel

- Aktion, die drei Zeichen einliest, das mittlere Zeichen ignoriert und das erste und dritte als Paar zurückgibt:

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
```

- Achtung: `return` erforderlich, um `IO` – Typ zurückzugeben!

… noch ein Beispiel:

```
echo3 :: Int -> IO ()
echo3 cnt = do
  putStr (show cnt ++ ": ")
  s <- getLine
  if s /= "" then do
      putStrLn $ show cnt ++ ": " ++ s
      echo3 (cnt + 1)
    else return ()
```

() =
Menge aller leeren Tupel,
"Einheitstyp" (*unit*);
vgl. **void** in anderen
Programmiersprachen

- Kombination von Kontrollstrukturen u. Aktionen
- Aktionen werden als Werte verwendet
- `do` – Notation geschachtelt

Kontrollstrukturen: `sequence`

```
sequence :: [IO a] -> IO [a]
```

- bekommt eine Liste von IO Aktionen
- liefert IO Aktion, die alle Aktionen nacheinander ausführt
- Rückgabe ist Liste der Resultate aller IO Aktionen
- Beispiel:

```
> sequence [getLine, getLine]
hallo
du
["hallo","du"]
```

Kontrollstrukturen: `sequence`

- Weiteres Beispiel:

```
> sequence $ map print [1,2,3]
1
2
3
[(),(),()]
```

- Warum Ausgabe von `[(),(),()]` ? – Liste der Rückgaben der Aktionen!
- Alternative:

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

```
> sequence_ $ map print [1,2,3]
1
2
3
```

Kontrollstrukturen: `mapM`

- Abkürzung für `sequence $ map`:

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

- bekommt eine Funktion und eine Liste
- führt `map` aus und sequenziert danach
- Beispiel:

```
> mapM print ["eins","zwei"]
"eins"
"zwei"
[(),()]
```

- `mapM_` analog zu `sequence_`

Noch einmal: das `Main` Modul

- Wichtig für ausführbare (kompilierte) Programme
  - Top-Level Modul
- Funktion `main` im Modul `Main` wird ausgeführt
- Jedes eigenständige Programm ist eine **Aktion**
  - Programme ändern den Zustand der Welt!
- Signatur: `main :: IO t` (für beliebigen Typ `t`)
- In der Regel einfach: `main :: IO ()`

I/O mit Dateien

Dateien in Haskell

- Dateipfad und –name: `type FilePath = String` (*standard prelude*)
- Dateien schreiben: `writeFile :: FilePath -> String -> IO ()`
  - Achtung: existierende Dateien werden überschrieben!
- Dateien erweitern: `appendFile :: FilePath -> String -> IO ()`
- Dateien lesen: `readFile :: FilePath -> IO String`
  - Lesen erfolgt verzögert ("*lazy I/O*"), obwohl I/O Aktion!

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

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

- Datei wird eingelesen in `String`
- Zeilen, Wörter, Zeichen zählen
- Ausgabe als `IO ()`

Fortgeschrittene Dateioperationen in Haskell

- `import System.IO`
- Verwaltung von I/O durch *Handles*
- I/O Modi *read*, *write*, *append*, *readWrite*
- Puffer-Modi ungepuffert, zeilen-, oder blockweise
- Such-Modi absolut, relativ, rückwärts

- siehe *Haskell 2010 Language Report*

Zufallszahlen in Haskell

Zufallszahlen

- Zufallszahlen erfordern eine Aktion! – Warum?
- `import System.Random`
- `randomRIO :: (a, a) -> IO a`
  - wie `randomR`, nutzt aber globalen Zufallsgenerator
- Beispiel: Aktion zufällig oft ausführen (mit gegebenem Maximum):

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

Beispiel: Zufallsstrings

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

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

```
> randomStr
"uafhknqnokhnolsizefeyoitkcsq"
```

Fallbeispiel: Hangman

(von Christoph Lüth)

Hangman (Wörter raten)

- Wort wird zufällig aus Wörterbuch ausgewählt
  - deutsch oder englisch
- Benutzer gibt Buchstaben ein
  - bereits eingegebene Buchstaben werden nicht mehr akzeptiert
- Wort wird maskiert ausgegeben
  - geratene Buchstaben werden angezeigt

Programmstruktur: Hauptfunktion `main`

```
main :: IO ()                                          ← siehe oben
main =
  catch (do                  ← Fehlerbehandlung (Control.Exception)
    as <- getArgs            ← Argumente aus Kommandozeile (System.Environment)
    let df= case as of
              ["de"] -> "dict-de_DE"
              ["en"] -> "dict-en_EN"
              _ -> "dict-de_DE"
    dict <- readFile df                                ← Wörterbuch einlesen
    putStrLn $ "Selected dictionary " ++ df ++ " (" ++
               show (length (lines dict)) ++ " entries)"
    secret <- pickRandom (filter ((5 <). length) (lines dict))
    play (map toLower secret) "" "")                   ← Funktion play aufrufen
    (\e-> putStrLn $ "Error: "++ show (e:: IOException))
```

Wort auswählen

Programmstruktur: Funktion `play`

```
play :: String -> String -> String -> IO ()
play word hits misses
  | all (`elem` hits) word = do
      putStrLn $ "Excellent --- you guessed "++ word++ "."
  | length misses == numTries = do
      putStrLn $ "The word is "++ word++ " -- you lose."
  | otherwise = do
      putStrLn (render hits word)
      putStrLn $ show (length misses) ++ " misses of " ++
                 show numTries ++ " tries."
      c <- getGuess hits misses
      if (not $ c `elem` word) then
        do putStrLn "Miss."; play word hits (c:misses)
        else play word (c:hits) misses
```

Wort

geratene Zeichen (positiv)

(negativ)

Ausgabe Buchstabe oder "_"

Zeicheneingabe durch Benutzer

Programmstruktur: Benutzereingabe

geratene Zeichen
(positiv) (negativ)

```
getGuess :: String -> String -> IO Char
getGuess hits misses = do
  putStr $ "Your guess? "
  i <- getLine
  case i of
    "?" -> do
      putStrLn $ "Your misses so far: "++ filter isAlpha misses
      getGuess hits misses
    [c] | c `elem` misses -> do
      putStrLn "You already guessed that."
      getGuess hits misses
    [c] -> return $ toLower c
    _ -> do
      putStrLn "Please, one guess at a time."
      getGuess hits misses
```

Zusammenfassung

- I/O in Haskell durch Aktionen
  - Seiteneffekte, keine referentielle Transparenz, Reihenfolge
- Elementare I/O Aktionen
- Komposition von I/O Aktionen und `do` – Notation
- `main` ist eine Aktion
  - jedes eigenständige Programm ist eine Aktion
- I/O mit Dateien
- Zufallszahlen benötigen auch Aktionen

nächstes Mal...

- Testen und Qualitätssicherung
  - Signaturen von ADTs
  - Eigenschaften und Axiome
  - Quickcheck
  - Fallbeispiele


Merry Xmas!