Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 20.10.09: Einführung und Rückblick

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Organisatorisches

- Vorlesung: Di 8 – 10 MZH 7250
- Übung: Do 10 – 12 MZH 7210 (nach Bedarf)
- Scheinkriterien:
  - 5 Übungsblätter
  - Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - Übungsgruppen 2 – 4 Mitglieder
  - Ggf. Fachgespräch am Ende

Warum?

- Haskell: Nicht nur für Übungsaufgaben
- Funktionale Sprachen sind Innovationsinkubatoren
- Funktionale Sprachen behandeln Zukunftsthemen heute

Themen

- Monaden und fortgeschrittene Typen
  - Was ist eine "Monade"?
  - `IO` und andere Monaden
  - Konstruktorklassen, Rang-2-Polymorphie
- Fortgeschrittene Datenstrukturen
  - Der Zipper
  - Ströme, Graphen, unendliche Datenstrukturen
- Nebenläufigkeit
  - Leichtgewichte Threads in Haskell
  - Queues, Semaphoren, Monitore ...
  - State Transactional Memory
- The Future of Programming
  - Domänenspezifische Sprachen (DSLs)
  - Fancy Types
  - The Next Big Thing: F#, Scala

Ressourcen

- Haskell-Webseite: `http://www.haskell.org/`
- Büchereien:
  - Haskell 98 Libraries
  - Haskell Hierarchical Libraries
- Compiler:
  - Glasgow Haskell Compiler (`ghc`) (Version 6.10)
  - `http://www.haskell.org/ghc/`
- Bücher und Artikel
  - Siehe `http://www.informatik.uni-bremen.de/~cxl/lehre/asp.ws09/`
  - Ab morgen!

Rückblick Haskell

- Definition von Funktionen:
  - lokale Definitionen mit `let` und `where`
  - Fallunterscheidung und guarded equations
  - Abseitsregel
  - Funktionen höherer Ordnung
- Typen:
  - Basisdatentypen: `Int`, `Integer`, `Rational`, `Double`, `Char`, `Bool`
  - Strukturierte Datentypen: `[a]`, `(a, b)`
  - Algebraische Datentypem: `data Maybe a = Just a – Nothing`

Rückblick Haskell

- Abstrakte Datentypen
- Module
- Typklassen
- Verzögerte Auswertung und unendliche Datentypen

I/O in funktionalen Sprachen

- Problem: Eingabe kann nicht als Funktion

  ```
  readLine ::() → String
  ```

  modelliert werden — zerstört referentielle Transparenz.

- Generelles Problem hier: Interaktion mit der Umwelt

- Mögliche Lösungen:

  - Seiteneffekte (e.g. Standard ML);
  - Continuations (Auswertungskontext explizit modellieren);
  - Streams: `readLine :: Instream-¿ (Instream, String)`
  - Einkapselung in Monaden (Haskell).

Monadische I/O

- Abstrakter Datentyp IO a :

```
(>>=)  :: IO t -> (t -> IO u) -> IO u  -- "then"
return :: t -> IO t                   -- "return"
```

- t :: IO a erst eine Aktion, gibt dann Wert vom Typ a zurück:

```
type IO a = World -> (a, World)
```

Monadische I/O

- Elementare Operationen:

```
getLine   :: IO String          -- eine Zeile lesen
putStr    :: String -> IO ()    -- Zeile ausgeben
putStrLn  :: String -> IO ()    -- Zeile mit LF ausgeben
```

- "Einmal I/O, immer I/O"
- Abhängigkeit von Umwelt am Typ erkennbar
- Daher:

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

Hauptprogramm hat keinen Rückgabewert, nur noch Interaktion.

Monadische I/O: Die do Notation

- Syntaktischer Zucker für Monaden:

```
echo =
  do s ← getLine              getLine
           putStrLn s ⟷      >>= λs → putStrLn s
           echo               >> echo
```

- Oder auch:

```
echo = do { s← getLine; putStrLn s; echo }
```

- Nützlich:

```
(>>) :: IO t → IO u → IO u
f >> g ≡ f >>= λ_ → g
```

Monadische I/O: Einfache Beispiele

```
echo :: IO ()
echo = getLine >>= putStrLn >> echo
echo = do { l <- getLine; putStrLn l; echo }

interactOnce :: (String -> String) -> IO ()
interactOnce f = getLine >>= (putStr . f)
interactOnce f = do { l<- getLine; putStrLn (f l) }

revecho :: IO ()
revecho = getLine >>= putStrLn . reverse >> revecho
revecho = do { l<- getLine; putStrLn (reverse l); reve
```

File I/O

Abstrakter Zugriff durch lazy evaluation:

```
type FilePath  = String
getContents    :: IO String
readFile       :: FilePath → IO String
writeFile      :: FilePath → String → IO ()
appendFile     :: FilePath → String → IO ()
```

Beispiel:

```
cntWords :: FilePath → IO ()
cntWords file = do c ← readFile file
                   let s = (length . words) c
                   putStr $ file ++ " : " ++ show s ++ "
```

Fortgeschrittene File I/O

```
data IOMode      = ReadMode | WriteMode | AppendMode
openFile         :: FilePath → IOMode → IO Handle

hGetContents     :: Handle → IO String   -- uvm.
hFlush           :: Handle → IO ()

hGetPosn         :: Handle → IO HandlePosn
hSetPosn         :: HandlePosn → IO ()

data SeekMode = AbsoluteSeek | RelativeSeek | SeekFrom
hSeek            :: Handle → SeekMode → Integer → IO
```

Weitere übliche Operationen (Buffering etc) siehe Haskell98 Library Report, Kap. 11.

Fehler!

Repräsentation durch den abstrakten Datentyp `IOError`.
Ausnahmebehandlung ähnlich in Java:

```
ioError  :: IOError → IO a
catch    :: IO a → (IOError → IO a)→ IO a
```

Beispiel:

```
cntW file = catch (cntWords file)
                  (λe→ putStr ("Error: " ++ (show
```

Analyse der Fehler durch `isDoesNotExistsError :: IOError-¿ Bool`
etc.

Kommandozeilenargumente

Interaktion mit der Umgebung: Modul `System`

```
data ExitCode = ExitSuccess | ExitFailure Int
getArgs       :: IO [String]
getProgName   :: IO String
getEnv        :: String → IO String
system        :: String → IO ExitCode
exitWith      :: ExitCode → IO a
```

Beispiel:

```
main = do r← getProgName; a← getArgs
          catch (mapM_ cntWords a)
                (λe→ putStrLn (r++ ": "++ (show e)))
```

Das Modul `Directory`

```
createDirectory                  :: FilePath → IO ()
removeDirectory, removeFile      :: FilePath → IO ()
renameDirectory, renameFile      :: FilePath → FilePath →
IO ()

getDirectoryContents  :: FilePath → IO [FilePath]
getCurrentDirectory   :: IO FilePath
setCurrentDirectory   :: FilePath → IO ()

data Permissions = ...
readable, writeable, executable, searchable :: Permis
getPermissions        :: FilePath → IO Permissions
setPermissions        :: FilePath → Permissions → IO
getModificationTime   :: FilePath → IO ClockTime
```

Das Modul Directory, Beispiel

```
import Directory
import Time
import System(getArgs)

cleanup dir =
 do now← getClockTime
    c← getDirectoryContents dir
    setCurrentDirectory dir
    mapM_ (λf → do {mt← getModificationTime f;
                    if ((last f ≡ '~') &&
                        tdDay (diffClockTimes mt now)
1)
                    then removeFile f else return ()})
main = do { d← getArgs; cleanup (head d) }
```

Systemfunktionen für Haskell

- Abstrakte Modellierung in Haskell98 Standard Library:
  `IO`, `Directory` `System`, `Time`
  Siehe Library Report

- Konkrete Modellierung in Modul Posix (nur für `GHC`) nach IEEE Standard 1003.1, e.g.:

```
executeFile :: FilePath                                  -- Command
            -> Bool                                      -- Search PATH?
            -> [String]                                  -- Arguments
            -> Maybe [(String, String)]                  -- Environment
            -> IO ()
```

More IO

Nützliche Kombinatoren (aus dem `Prelude`):

```
sequence   :: [IO a] → IO [a]
sequence_  :: [IO a] → IO ()
mapM       :: (a → IO b) → [a] → IO [b]
mapM_      :: (a → IO b) → [a] → IO ()
```

Mehr im Modul `Monad` (Library Report, Kapt. 10).

Zusammenfassung

- Abhängigkeit von Aussenwelt in Typ *IO* kenntlich
- Benutzung von IO: vordefinierte Funktionen in der Haskell98 Bücherei
- Nächstes Mal:
  - Was steckt dahinter?
  - Flucht aus Alcatraz – IO für Erwachsene
  - Endlich Variablen