Praktische Informatik 3

Christian Maeder

WS 03/04

Vorlesung vom 12.1.2004: Ein/Ausgabe in funktionalen Sprachen

Inhalt

- Wo ist das Problem?
- Aktionen und der Datentyp `IO`.
- Vordefinierte Aktionen
- Aktionen als Werte
- Monaden und Anwendungen

Ein- und Ausgabe in funktionalen Sprachen

- **Problem:**

  Funktionen mit Seiteneffekten sind nicht referenziell transparent (bzw. rein), d.h. die Ausgabe hängt nicht nur von der Eingabe ab. Beispiele:

  - Zufallszahlen
  - `Time.getClockTime :: ?? ClockTime`
  - `getLine :: ?? String` (in ML: `() -> String`)
  - `putStrLn :: String -> ?? ()`

- **Lösung:**

  Seiteneffekte durch den Typkonstruktor `IO` beim Ergebnis kennzeichnen!

  - `putStrLn :: String -> IO ()`

- **Lösung:**

  Seiteneffekte durch den Typkonstruktor `IO` beim Ergebnis kennzeichnen!

  - `putStrLn :: String -> IO ()`

```
hello :: IO ()
hello = putStrLn "Hello World!"
```

- `Time.getClockTime :: IO ClockTime`
- `Random.randomIO :: Random a => IO a`
- `getLine :: IO String`

- `Time.getClockTime :: IO ClockTime`
- `Random.randomIO :: Random a => IO a`
- `getLine :: IO String`

`IO` ist abstrakter Datentyp, der das Ergebnis ‘`a`’ einer Aktion kapselt.

- **Lösungsfortsetzung:** Zugriff auf Werte und Komposition von Aktionen per ‘do’-Syntax

```
exp -> ... | do { stmts }

stmt -> pat <- exp | exp
```

- **Lösungsfortsetzung:** Zugriff auf Werte und Komposition von Aktionen per ‘`do`’-Syntax

  ```
  exp -> ... | do { stmts }
  ```

  ```
  stmt -> pat <- exp | exp
  ```

  - vergleichbar mit imperativen Zuweisungen (single assignment)
  - Variablen (in `pat`) sind neu und dürfen danach in Ausdrücken (ggf. mehrfach) vorkommen
  - rechter Ausdruck vom Typ ‘`IO a`’, linkes Pattern vom Typ ‘`a`’
  - Für mehrere Zuweisungen gilt die Abseitsregel
  - letzter Ausdruck (ohne Pattern!) bestimmt Ergebnistyp

- **Beispiele:** Komposition von Aktionen

```
getAndPut :: IO ()
getAndPut = do str <- getLine
               putStrLn str
```

- **Beispiele:** Komposition von Aktionen

```
getAndPut :: IO ()
getAndPut = do str <- getLine
               putStrLn str
```

- rekursive Komposition

```
echo :: IO ()
echo = do getAndPut
          echo
```

- mit Aufruf einer reinen Funktion (reverse):

```
ohce :: IO ()
ohce = do str <- getLine
          putStrLn (reverse str)
          ohce
```

- mit Aufruf einer reinen Funktion (reverse):

```
ohce :: IO ()
ohce = do str <- getLine
          putStrLn (reverse str)
          ohce
```

  - reine Funktionen gezielt und streng getrennt von Aktionen verwenden

- mit Aufruf einer reinen Funktion (reverse):

```
ohce :: IO ()
ohce = do str <- getLine
          putStrLn (reverse str)
          ohce
```

  - reine Funktionen gezielt und streng getrennt von Aktionen verwenden

  - Rekursion und Fallunterscheidungen ermöglichen Dialoge

- **Beispiel:** Fallunterscheidung und `return`

```
echo2 = do str <- getLine
           if str /= "quit"
             then do putStrLn str   -- abseits!
                     echo2
             else return ()
```

- **Beispiel:** Fallunterscheidung und `return`

```
echo2 = do str <- getLine
           if str /= "quit"
             then do putStrLn str   -- abseits!
                     echo2
             else return ()
```

  - `return :: a -> IO a`
  - Ein reiner Wert kann (durch `return`) in eine Aktion verwandelt werden, aber nicht umgekehrt
  - einmal IO, immer IO!

Vordefinierte Aktionen

```
fail :: String -> IO a
fail = error

print :: Show a => a -> IO ()
print = putStrLn . show
```

Vordefinierte Aktionen

```
fail :: String -> IO a
fail = error

print :: Show a => a -> IO ()
print = putStrLn . show
```

- Ganze Textdateien (verzögert) lesen und schreiben:

```
type FilePath =  String
readFile      ::  FilePath               -> IO String
writeFile     ::  FilePath -> String -> IO ()
```

Aktionen als Argumente

- Liste von Aktionen nacheinander ausführen:

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

- Zwei Aktionen ausführen

```
(>>=) :: IO a -> (a -> IO b) -> IO b    -- bind
c >>= f = do x <- c
             f x                        -- kein return!
```

- Zwei Aktionen ausführen

```
(>>=) :: IO a -> (a -> IO b) -> IO b    -- bind
c >>= f = do x <- c
             f x                        -- kein return!
```

- Spezialfall: zweite Aktion ist unabhängig von der ersten

```
(>>) :: IO a -> IO b -> IO b
a >> b =
          a >>= \ _ -> b    -- oder    do a
                                       b
```

- Die `do`-Notation ist syntaktischer Zucker für Applikationen von `(>>)` und `(>>=)`!

```
c >>= \x -> f x
```

$\Longleftrightarrow$

```
do x <- c
   f x
```

- Die do-Notation ist syntaktischer Zucker für Applikationen von (>>) und (>>=)!

```
c >>= \x -> f x
```

$\Longleftrightarrow$

```
do x <- c
   f x
```

- Notationsalternativen ($\lambda x.f(x) = f$)

```
getAndPut = getLine >>= putStrLn
```

```
echo = getAndPut >> echo
```

- I.a. ist die do-Notation lesbarer

Weitere Funktionen für Aktionen

- Spezialfall für Typ `()` in `sequence`:

```
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (c:cs) = c >> sequence_ cs
```

- Nur die Seiteneffekte der Aktionen sind interessant

- 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
```

Eigene Kontrollstrukturen

- z.B. While-Schleife

```
while :: IO Bool -> IO () -> IO ()
while cond act =
    do b <- cond
       if b then do act
                    while cond act
            else return ()
```

Eigene Kontrollstrukturen

- z.B. While-Schleife

```
while :: IO Bool -> IO () -> IO ()
while cond act =
    do b <- cond
       if b then do act
                    while cond act
            else return ()
```

- wird wegen Rekursion kaum benutzt

Monaden (und ihre Verbindung zu IO)

- Eine (Typ-)Konstruktorklasse mit einer Typkonstruktor(-variablen) ‘m’

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b    -- Spezialfall
    fail   :: String -> m a
```

Monaden (und ihre Verbindung zu IO)

- Eine (Typ-)Konstruktorklasse mit einer Typkonstruktor(-variablen) ‘m’

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b    -- Spezialfall
    fail   :: String -> m a
```

- IO ist eine Instanz von Monad (m a $\rightsquigarrow$ IO a)
- ‘m a’ ist “Berechnung” von Werten des Typs a

- Eigenschaften von Monaden
  - Kleisli-Komposition

```
(>@>) :: Monad m => (a -> m b) ->
                    (b -> m c) ->
                    (a -> m c)
(f >@> g) x = f x >>= g              -- bind
```

  - `return` ist neutrales Element

```
return >@> f   =   f >@> return   =   f
```

  - `>@>` ist assoziativ

```
f >@> (g >@> h)   =   (f >@> g) >@> h
```

- "Berechnungsfunktionen" bilden Halbgruppe (Monoid)

- Eine weitere Monadeninstanz (Exkurs)
  - Der Typkonstruktor für Listen (`m a` $\rightsquigarrow$ `[a]`)

```
return x = [x]
xs >>= f = concat (map f xs)
fail _ = []
```

  - `(>>=) :: [a] -> (a -> [b]) -> [b]`

- Eine weitere Monadeninstanz (Exkurs)
  - Der Typkonstruktor für Listen (`m a` $\rightsquigarrow$ `[a]`)

```
return x = [x]
xs >>= f = concat (map f xs)
fail _ = []
```

  - `(>>=) :: [a] -> (a -> [b]) -> [b]`
  - mehrere (reine) Ergebnisse parallel (als Liste) verwalten
  - keine Seiteneffekte bei dieser Instanz!

- Listenumschreibungen mit mehreren Generatoren

```
[ (x, y) | x <- l1, y <- l2 ]
    =
do x <- l1
   y <- l2
   return (x, y)
```

- Listenumschreibungen mit mehreren Generatoren

```
[ (x, y) | x <- l1, y <- l2 ]
    =
do x <- l1
   y <- l2
   return (x, y)
```

```
is :: [Int]  -- 0 - 99
is = do x <- [0 .. 9]
        y <- [0 .. 9]
        return (10 * x + y)
```

IO Beispiel: zufälligen String erzeugen

```
randStr :: Int -> IO String
randStr n =
  if n <= 0 then return "" else
  do c <- randomIO
     if isAlpha c
       then do
         cs <- randStr (n-1)
         return (c:cs)
       else randStr n

main = randStr 10 >>= putStrLn
```

- Das (standalone) Hauptprogramm (`main`) ist eine Aktion.
- Aktion mit Seiteneffekten wird vom Laufzeitsystem interpretiert/ausgeführt.
- Aktion ist mit (abstraktem) Shell-Skript vergleichbar, dessen Text rein funktional per Stringkonkatentation erstellt wurde.
- Seiteneffekte durch vordefinierte Aktionen.
- Für Korrektheit, IO (durch `return`) möglichst vermeiden.
- Besser als imperative (oder Skript-) Sprachen durch: strenge Typisierung, Polymorphie, Fktn. höherer Ordnung

- Es gibt weitere IO-Bibliotheken, die hier nicht vorgestellt werden:
  - Fehlerbehandlung mit `throw` und `catch` von `Exceptions`

- Es gibt weitere IO-Bibliotheken, die hier nicht vorgestellt werden:
  - Fehlerbehandlung mit `throw` und `catch` von `Exceptions`
  - Parallelverarbeitung durch Threads (`Control.Concurrent`) oder Prozesse (`System.Posix`)

- Es gibt weitere IO-Bibliotheken, die hier nicht vorgestellt werden:
  - Fehlerbehandlung mit `throw` und `catch` von `Exceptions`
  - Parallelverarbeitung durch Threads (`Control.Concurrent`) oder Prozesse (`System.Posix`)
  - Graphische Benutzerschnittstellen (GUIs), etc.

- Es gibt weitere IO-Bibliotheken, die hier nicht vorgestellt werden:
  - Fehlerbehandlung mit `throw` und `catch` von `Exceptions`
  - Parallelverarbeitung durch Threads (`Control.Concurrent`) oder Prozesse (`System.Posix`)
  - Graphische Benutzerschnittstellen (GUIs), etc.

- Unreine Funktion aus `Debug.Trace`

```
trace :: String -> a -> a
```

Zusammenfassung

- Ein/Ausgabe in Haskell durch Aktionen per ‘`do`’
- Aktion (Typ `IO a`) ist i.a. seiteneffektbehaftet
- Standardaktionen: (Konsole zeilenweise) `getLine`, `putStrLn`; (Textdateien) `readFile`, `writeFile`
- Aktionen als Argumente, `do`-Notation basiert auf bind:

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

- Die Typkonstruktorklasse `Monad`, mit `(>>=)`,`(>>)`, `return` und `fail`.
- Listen als Monadeninstanz (Listenumschreibung per ‘`do`’)