Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag fünf - geschafft

Stephan Mielke, 26.01.2015

Technische Universität Braunschweig, IPS

Überblick

- **Das Array**
- Monaden
- IO

Das Array

- In Haskell existieren nicht nur Listen zur Speicherung und Verarbeitung von Daten sondern auch zwei Array Formen
- Arrays in Haskell besitzen immer eine feste Größe die bei der Erstellung angegeben wird

Statische Arrays

- Es muss das Modul `Data.Array.IArray` importiert werden
- Die einzelnen Elemente eines Arrays werden mit dem `!` Operator angesprochen
  z.B. `a!5` gibt das Element mit dem Index 5 aus dem Array `a` wieder

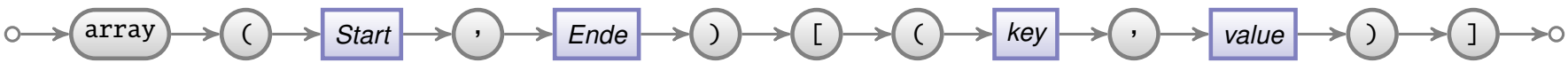
Statische Arrays

- Es muss das Modul `Data.Array.IArray` importiert werden
- Die einzelnen Elemente eines Arrays werden mit dem `!` Operator angesprochen
  z.B. `a!5` gibt das Element mit dem Index 5 aus dem Array `a` wieder

Als Array Index kann jeder Datentyp verwendet werden, welcher die Typklasse `Ix` implementiert

Statische Arrays

- Es muss das Modul `Data.Array.IArray` importiert werden
- Die einzelnen Elemente eines Arrays werden mit dem `!` Operator angesprochen
  z.B. `a!5` gibt das Element mit dem Index 5 aus dem Array `a` wieder



In diesem Fall wird dem Array eine Liste mit Tupeln übergeben bei dem das erste Element der „Primarykey“ ist (wie eine Map)

Statische Arrays

```
listArray :: (Ix i, IArray a e) => (i, i)
    -> [e] -> a i e
myArray1 = (listArray ('a','e') [10..15])
    :: Array Char Int
```

Statische Arrays

```
listArray :: (Ix i, IArray a e) => (i, i)
    -> [e] -> a i e
myArray1 = (listArray ('a','e') [10..15])
    :: Array Char Int
```

```
array :: (Ix i, IArray a e) => (i, i)
    -> [(i, e)] -> a i e
myArray2 = (array (1,5) [(k,k*2)|
    k <- [1..5]):: Array Int Int
```

Achtung

Die Anzahl der Listen Elemente und der Platz müssen nicht übereinstimmen, solange das Array nicht ausgegeben (`show a`) wird.

Statische Arrays

```
accumArray :: (IX i, IArray a e) => (e -> e' -> e)
   -> e -> (i,i) -> [(i, e')] -> a i e

myArray3 = (accumArray (+) 0 (0,4) [(i `mod` 5
   , 1) | i <- [1..123]]) :: Array Int Int
```

Statische Arrays

```
accumArray :: (IX i, IArray a e) => (e -> e' -> e)
   -> e -> (i,i) -> [(i, e')] -> a i e

myArray3 = (accumArray (+) 0 (0,4) [(i `mod` 5
   , 1) | i <- [1..123]]) :: Array Int Int
```

```
array (0,4)[(0,24),(1,25),(2,25),(3,25),(4,24)]
```

Wichtige Array Funktionen

- `amap` ist die Array-Variante der
- `map` Funktion für Listen
- `elems` wandelt das Array in eine Liste um (nur die Werte)
- `assocs` wandelt das Array in eine Liste von Tupeln der Form (k, v) um
- Der Operator `\\` (update) ändert in einem Array die gegebenen Wertpaare.

Statische vs. dynamische Arrays

- Bei einem Update mit dem \\ Operator (update) wird bei statischen das gesamte Array kopiert und die Änderungen vorgenommen
- Somit dauert es bei statischen länger als bei dynamischen

Dynamische Arrays

- Import von `Data.Array.Diff`
- Funktionen heißen gleich nur Typ ist `DiffArray` statt `Array`
- Besitzen zwar eine Konstante Zeit beim Update
- Aber erhöhte Zugriffszeit beim Lesen
- Durch geschickte Array Konstruktion kann jedoch fast vollständig auf Updates verzichtet werden

Haskell API

Für weitere Datentypen und deren Funktionen siehe:
`haskell.org/hoogle`

Überblick

- Das Array
- **Monaden**
- IO

Monaden

- Monaden sind ein mathematisches Konzept aus der Kategorientheorie
- Werden eingesetzt um Funktionen miteinander zu kombinieren
- Ist in Haskell eine polymorphe Datenstruktur mit speziellen Funktionen
- Das Prinzip ist:
  - Sequenzialisierung gemäß des Continuation-Style Programming der Kontrollfluss kehrt nicht zum Aufrufer zurück sondern geht zur Nachfolgefunktion
  - Darstellung und Transformation eines versteckten Zustands (Hiding)
  - Sicherung von Single-Threadedness dadurch, weil keine dagegen verstoßende Funktion benutzt werden kann

Monaden - Klasse

```
class  Monad m  where
    -- verbinden zweiter Funktionen
    -- Ergebnis ist Argument der zweiten Funktion
    (>>=)    :: forall a b. m a -> (a -> m b) -> m b
    -- verbindet zwei Funktionen aber verwirft jedes
    -- Ergebnis (wie in Imperativen Sprachen)
    (>>)     :: forall a b. m a -> m b -> m b
    m >> k   = m >>= \_ -> k
    -- fuegt einen Wert in den Monaden Typ ein
    return   :: a -> m a
    -- gibt eine Fehlernachricht zurueck
    fail     :: String -> m a
    fail     = error
```

Monaden - Klasse

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = case mA of
    Nothing -> Nothing
    Just a     -> case mB of
                    Nothing -> Nothing
                    Just b  -> Just (a + b)
```

Monaden - Klasse

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = case mA of
    Nothing -> Nothing
    Just a     -> case mB of
                    Nothing -> Nothing
                    Just b  -> Just (a + b)
```

```
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
               mB >>= (\b ->
                 return (a + b)))
```

Do-Notation

- Mit der `do`-Notation werden Monaden (>>=) zusammengefasst pro Zeile
- Somit ist es syntaktischer Zucker
- Für die Verwendung der `do`-Notation sind 4 Regeln zu beachten

Do-Notation - Regel 1

- Einzelne Anweisungen benötigen keine Umformung.
- Das `do` wird einfach weggelassen.

```
do
  e
```

Do-Notation - Regel 1

- Einzelne Anweisungen benötigen keine Umformung.
- Das `do` wird einfach weggelassen.

```
do
  e
```

```
e
```

Do-Notation - Regel 2

- Wird der Rückgabewert nicht benötigt
- Dann wird die Anweisung nach vorne gezogen

```
do
  e
  <Anweisung>
```

Do-Notation - Regel 2

- Wird der Rückgabewert nicht benötigt
- Dann wird die Anweisung nach vorne gezogen

```
do
  e
  <Anweisung>
```

```
e >>= \_ -> do
  <Anweisungen>
```

Do-Notation - Regel 3

- Wird der Rückgabewert mit Pattern-Matching ausgewertet
- Dann muss eine Hilfsfunktion dies übernehmen

```
do
  pattern <- e
  <Anweisungen>
```

Do-Notation - Regel 3

- Wird der Rückgabewert mit Pattern-Matching ausgewertet
- Dann muss eine Hilfsfunktion dies übernehmen

```
do
  pattern <- e
  <Anweisungen>
```

```
let ok pattern = do
    <Anweisungen>
  ok _ = fail "Fehler"
in e >>= ok
```

Do-Notation - Regel 4

- Wird ein Wert mit `let` gespeichert, kann dies vor das `do` gezogen werden
- Das `in` ist im `do`-Block optional

```
do
  let <Deklaration>
  in <Anweisungen>
```

Do-Notation - Regel 4

- Wird ein Wert mit `let` gespeichert, kann dies vor das `do` gezogen werden
- Das `in` ist im `do`-Block optional

```
do
  let <Deklaration>
  in <Anweisungen>
```

```
let <Deklaration>
in do
      <Anweisungen>
```

Do-Notation - If-Then-Else

Wir erwarten

```
f = do
  if <irgendwas> then
    <Anweisungen>
  else
    <Anweisungen>
```

Aber!

```
f =
  if <irgendwas> then do
    <Anweisungen>
  else do
    <Anweisungen>
```

Do-Notation - Beispiel

```
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
                mB >>= (\b ->
                  return (a + b)))
```

Do-Notation - Beispiel

```
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
                mB >>= (\b ->
                  return (a + b)))
```

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = do
  a <- mA
  b <- mB
  return (a + b)
```

Vordefinierte Monaden

- Writer - für Debug / Logging / Tracing
- Reader - zum Lesen von gemeinsamen Zuständen (global)
- State - Verknüpfung von Writer und Reader für gemeinsame Zustände für zustandsbasierte Rechnungen

Überblick

- Das Array
- Monaden
- IO

Hangman I

```
import Data.Char
import System.IO

w = "Lambda"
maxI = 5

hangman :: String -> Int -> IO ()
hangman cs i | i > maxI = putStrLn "\nverloren"
             | all (`elem` cs) (map toLower w) =
               putStrLn "\ngewonnen"
```

Hangman II

```
hangman cs i =
  do
    putStrLn "␣"
    printWord cs
    putStrLn "\nWelcher␣Buchstabe?"
    c <- getChar >>= (return.toLower)
    if (c `elem` (map toLower w)) then
      hangman (c:cs) i
    else do
      putStrLn $ "\n" ++ (show (i+1)) ++ "␣falsch\n"
      hangman cs (i+1)
```

Hangman III

```
printWord :: String -> IO ()
printWord cs = mapM_ pC w
  where
    pC x  | toLower x `elem` cs = putChar x
          | otherwise = putChar '_'


main = do
  hSetBuffering stdin NoBuffering
  hangman "␣" 0
```

Das O in IO

- `print :: Show a =>a -> IO()`
  gibt jeden Datentyp der `Show` implementiert aus
- `putChar :: Char -> IO()`
  gibt ein `Char` aus
- `putStr :: String -> IO()`
  gibt einen `String` aus

  ```
  putStr = sequence_map putChar
  ```

Das O in IO

- `writeFile :: FilePath -> String -> IO()`
- `type Filepath = String`
- Schreibt den `String` mittels Stream in eine Datei

Das I in IO

- `readLn :: Read a =>IO a`
  liest jeden Datentyp der `Read` implementiert ein
- `getChar :: IO Char`
  liest ein Char ein
- `getLine :: IO String`
  liest die ganze Zeile ein als String
- Die Pufferung der Eingaben ist über `hSetBuffering` einstellbar
  für Windows bekommt es der GHC trotzdem nicht hin :(

Das I in IO

- `readFile :: FilePath -> IO String`
- `type Filepath = String`
- Liest die Datei als String ein

Wortsuche in einer Datei

```
isInfix :: String -> String -> Maybe String
isInfix [] _ = Nothing
isInfix t w | w == take (length w) t = Just t
            | otherwise = isInfix (tail t) w
```

Wortsuche in einer Datei

```
main = do
  putStrLn "Dateipfad: "
  filepath <- getLine
  putStrLn "gesuchtes Wort: "
  w <- getLine
  c <- readFile filepath
  case (isInfix c w) of
    Nothing -> putStrLn "nicht enthalten"
    Just s -> putStrLn $
        "an Stelle " ++ (take 100 s)
```

Zusammenfassung

Ab jetzt seid Ihr auf dem gleichen Wissenstand, auf dem ich bin.

- Nun seid ihr dran
  - Stellt Fragen
  - Schlagt auf `www.haskell.org/hoogle` nach
  - Oder vergesst alles schnell wieder