Funktionale Programmierung in Haskell

Unendliche Strukturen

Robert Steuck (inf8948)
Philipp Pribbernow (inf8949)

Unendliche Strukturen

Definition

- Prinzipiell Datenstrukturen wie Listen, Bäume etc.
- Aber: endlos rekursive Beschreibungsvorschrift => unendliche Länge

Vorteile

- technische / anwendungsspezifische Details irrelevant (z.B. Speichergrenzen)
- einfache Definition (Keine Rekursionsabbruchbedingung, weniger Parameter)
- Bei Verwendung wird erst entschieden wie weit die Definition ausgewertet werden soll

Lazy Evaluation

- In Haskell werden Ausdrücke nicht strikt ausgewertet ...
- ... sondern erst wenn diese zur Berechnung "benötigt" werden
- Einfaches Beispiel
- Konstantenfunktion

```haskell
const1 x = 1
```

- Parameter x wird nicht benötigt => nicht ausgewertet:
- Hinweis: Strikte Auswertung (Call-by-Value) möglich mit **($!)**
- Lazy Evaluation & Call-by-Value

```
> const1 $ 1 `div` 0
> 1  -- und nicht *** Exception: divide by zero!

> ($!) const1 $ 1 `div` 0
> *** Exception: divide by zero!
```

- Ein weiteres Beispiel
- Eager-Auswertung files/lazy.hs

```
square (7 + 3)
= square (10)
= square 10 * 10
= 100
```

- Lazy-Auswertung files/lazy.hs

```
square (7 + 3)
= x * x where x = 7 + 3
= x * x where x = 10
= 10 * 10
= 100
```

- Lazy Evaluation ermöglicht das Auswerten unendlicher Strukturen "nach Bedarf"
- d.h.: Es werden nur (rekursive) Aufrufe durchgeführt, die zur Berechnung des Ergebnisses nötig sind:
- Berechnung Integerliste files/lazy.hs

```
intlist :: Integer -> [Integer]
intlist i = i : intlist (i+1)

--oder

ones = 1 : ones

> [1,1,1,1,1,1,1.....
```

Unendliche Listen

Definition

- `data [a] = [] | a : [a]`

- Unendliche Listen werden ohne "obere Grenze" erstellt:
- `[1..]`
  `> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,{Interrupted}`

- andere Erzeugungsmöglichkeiten: 2 Parameter P1 und P2
- P1 = Startposition
- P2 - P1 = Schrittweite
- Funktionen auf unendlichen Listen files/lazy.hs

```
[1,2..] > [1,2,3,4,5,6,...
```

```
[2,1..] > [2,1,0,-1,-2,-3...

[1,1..] > [1,1,1,1,1,      ≡  ones = 1 : ones
```

- Unendliche Listen Können als Parameter für Funktionen dienen wie endliche Listen ...
- ... die Resultate können trotzdem endlich sein
- Beispiele:
- Funktionen auf unendlichen Listen files/lazy.hs

```
head [n..] = n
take n [1..] = [1..n]
[m..]!!n = m + n
```

Folgen und Mengen

Folge der natürlichen Zahlen

- Natürliche Zahlenfolge files/FolgenUndGrenzwerte.hs

```
nat :: [Integer]
nat = nachfolger 0
        where nachfolger n = n : (nachfolger (n+1))
```

Fibbonacci-Folge

- kurze Version O(fib n): fib 30 => (5.44 secs, 277164136 bytes)
- Fibonnaci 1. Entwurf files/FolgenUndGrenzwerte.hs

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- lange Version O(n): fib 30 => (0.00 secs, 0 bytes)
- Fibonnaci 2. Entwurf files/FolgenUndGrenzwerte.hs

```
-- Fibonacci linear
fib2 :: Integer -> Integer
fib2 x  = case x of
          0 -> 0
          1 -> 1
          otherwise -> fib2' 0 1 (x-2)
          where fib2' a b n | n == 0 = res
                      | otherwise = fib2' b res (n-1)
                      where res = a + b
```

- unendliche Version O(n): fibs !! 30 => (0.00 secs, 0 bytes)
- Fibonnaci 3. Entwurf files/FolgenUndGrenzwerte.hs

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Mengendarstellung

- Zur Darstellung von unendlichen Mengen, z.B.
- {x | x² < 10}

- Problem: List Comprehension funktioniert nicht wie bei endlichen Listen, denn ...
- [x | x <- [0..], square x < 10]

- ... führt zur Endlosrekursion: > 0,1,4,9,⊥
- Erklärung:
- List comprehension ist äquivalent zu diesem Ausdruck:
- Comprehension mit Filter und Map files/example2.hs

```
filter (<10) (map square [0..])

---------------------------------------------

filter :: (a -> Bool) -> [a] -> [a]
filter _ []                  = []
filter p (x:xs) | p x        = x : filter p xs
                | otherwise = filter p xs
```

- Map und Filter arbeiten auf endliche Listen und wollen diese ganz auswerten
- Lösung: Ausnutzen von Lazy Evaluation:
- Lazy Evaluation mit TakeWhile files/example3.hs

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []                                -- nur für endliche
Liste relevant
takeWhile p (x:xs)      | p x = x : takeWhile p xs -- Lazy: takeWhile
                        | otherwise = []
```

- Beispiel: Sieb des Eratosthenes
- Sieb des Erastosthenes files/example4.hs

```
primes :: [Int]

primes       = sieve [2..]
sieve (x:xs) = x : sieve [y |y <- xs , y `mod` x > 0]
```

Anwendungen

- Finde alle Indizes eines Elements:
- FindIndices files/beispiele.hs

```
findIndices :: (a -> Bool) -> [a] -> [Int]
findIndices p xs =  [ i | (x,i) <- zip xs [0..], p x ]
```

- FindIndex files/beispiele.hs

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
findIndex p =  listToMaybe . findIndices p
```

- Finde Element an einem bestimmten Index:
- FindIndex files/beispiele.hs

```
elemAt :: Int -> [a] -> Maybe a
elemAt ix xs = listToMaybe [a | (i,a) <- zip [0..] xs, i == ix] -- Lazy
Evaluation
                                                                  -- mit
head
--wie (!!), welche bei [] aber eine Exception wirft
```

Listen als Grenzwerte

- In der Mathematik werden unendliche Objekte als Grenzwert unendlicher Sequenzen approximiert
- z.B. PI = 3,14159.... = Grenzwert der Sequenz: 3, 3.1, 3.14, 3.141, 3.1415....
- Jeder Wert dieser Sequenz ist eine Approximation von PI
- Unendliche Listen können auch approximiert werden:
- `⊥, 1:⊥, 1:2:⊥, 1:2:3:⊥ ...`

Ordnungen über Approximationen

- Reflexivität: $x \subseteq x$
- Transitivität: $(x \subseteq y) \wedge (y \subseteq z) \Rightarrow (x \subseteq z)$
- Antisymmetrie: $(x \subseteq y) \wedge (y \subseteq x) \Rightarrow (x = y)$

Für alle Zahlen, Booleans, Character und Aufzählungstypen jeglicher Art gilt:

- $x \subseteq y \equiv (x = \bot) \vee (x = y)$

Für Listen gilt:

- $\bot \subseteq xs$
- $[] \subseteq xs \equiv xs = []$
- $(x:xs) \subseteq (y:ys) \equiv (x \subseteq y) \wedge (xs \subseteq ys)$

Beispiel:

- `[ 1 , ⊥ , 3 ] ⊆ [ 1 , 2 , 3 ] und`
  `[ 1 , 2 , ⊥ ] ⊆ [ 1 , 2 , 3 ]`

Erweiterung der Approximationsordnung

- Jede Approximationskette $x_0 \subseteq x_1 \subseteq$ ... usw. besitzt einen Grenzwert
- Grenzwert $\lim n \to \infty\ x_n$ erfüllt folgende Bedingungen:
- 1) $x_k \subseteq \lim n \to \infty\ x_n$ für alle k
- 2) $x_k \subseteq y$ für alle k => $\lim n \to \infty\ x_n \subseteq y$

Eigenschaften unendlicher Listen

- Prinzip der Induktion reicht nicht aus um jede Eigenschaft von unendlichen Liste zu beweisen

- Beispiel Funktion Iterate

  ```
  iterate  :: (α -> α) -> α -> [α]
  iterate f x = x : iterate f (f x) -- erzeugt eine unendliche Liste
  ```

- Folgende Behauptung soll gezeigt werden:
- `iterate f x = x : map f (iterate f x)`

- Aber: kein geeignetes Argument für Induktion
- Alternative ??: Beweisen dass die Elemente der beiden Listen an den gleichen Indizes identisch sind:
- `xs !! n = ys !! n`

- Nein, denn für : xs = ⊥ und ys = [⊥] liefern alle indizierten Zugriffe den gleichen Wert: ⊥
- Lösung:
- Es wird eine Hilfsfunktion *approx* zum Approximieren von unendlichen Listen benötigt
- ... diese vergleicht die **Approximationen** zweier unendlicher Listen
- Appoximationsfunktion files/FolgenUndGrenzwerte.hs

  ```
  approx   :: Integer -> [α] -> [α]
  approx (n+1) [] = []
  approx (n+1) (x:xs) = x : approx n xs

  -- Hinweis: approx (n) (x:xs) = x : approx (n-1) xs ist nicht identisch
  ```

- approx n xs = approx n ys
  =>
  xs = ys
- Beweise xs ⊆ ys durch (approx n xs) ⊆ (approx n ys) für alle n gilt.
- Behauptung: iterate f (f x) = map f (iterate f x)
- Einsetzen: approx n (iterate f (f x)) = approx n (map f (iterate f x))

Beweis an der Tafel !

Zyklische Strukturen

Definition

- Datenstruktur mit zyklischen Graphen

Beispiele

Unedliche Liste mit rekursiver definition:

Antworten files/zyklisch.hs

```
answers :: [Int]
answers = 42 : answers
answers = 42 : answers
answers = 42 : 42 : answers
answers = 42 : 42 : 42 : answers
...
```

Graph:

Funktioniert auch mit Strings:

Unendlicher String files/zyklisch.hs

```
tooMuch :: String
tooMuch = "You know " ++ muchMore
                    where muchMore = "too much, " ++ muchMore
```

- Graph:
- Beide Beispiele haben konstanten Speicherbedarf

Alternative Erzeugung durch repeat

Definition files/zyklisch.hs

```
repeat x :: a -> [a]
repeat x = x : reapeat x

answers = repeat 42
```

- kein zyklischer Graph!

```
repeat 42
 {- wird ersetzte durch -}
42 : repeat 42
```

- besser:

Bessere Definition files/zyklisch.hs

```
repeat x :: a -> [a]
repeat x = xs
           where
             xs = x : xs
```

Effizienzvorteil zyklischer Strukturen

Verbesserung von Iterate files/zyklisch.hs

```
iterate  :: (a -> a) -> a -> [a]

{- Definition ohne map O(n) -}
iterate f x = x : iterate f (f x)

{- Definition mit map O(n²) -}
iterate f x = x : map f (iterate f x)

{- zyklische Definition O(n) -}
iterate f x = xs
              where
                xs = x : map f xs
```

Unendliche Bäume

Definition

- Unendlicher Baum files/Trees.hs

```
data Baum a = Knoten a [Baum a]
```

- Beliebig viele Kindknoten
- Blatt:
- `Knoten a []`

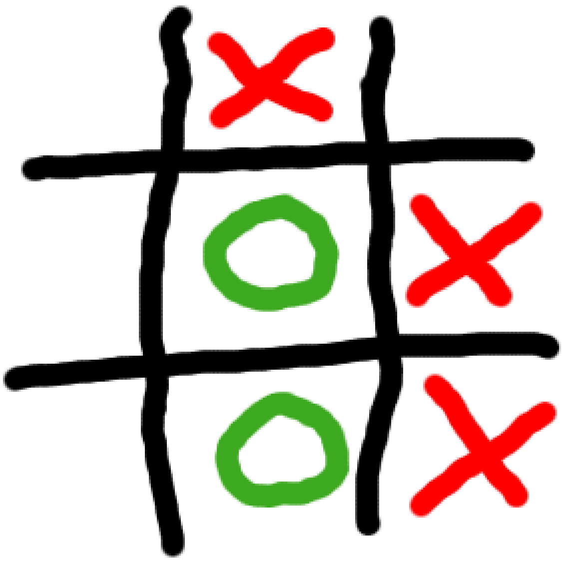
Erzeugung

- Unendlicher Baum files/Trees.hs

```
mkBaum :: (a -> [a]) -> a -> (Baum a)         -- (a -> [a]) generiert die Kinder
mkBaum f x = Knoten x (map (mkBaum f) (f x)) -- unendliche Definition
```

- Die Kinder werden mit folgender Funktion erzeugt
- `genKinder :: a -> [a]`

Beispiel Tic Tac Toe



- Konstruktion eines unendlichen Baumes für Tic Tac Toe
- Datendefinitionen files/Trees.hs

```
data Farbe = Leer | Schwarz | Weiss
data Position = Pos { amZug :: Farbe, brett :: [Farbe]}

startPosition = Pos Weiss (replicate 9 Leer) -- Ausgangspunkt der Berechnung
```

- Zugfunktion files/Trees.hs

```
moeglicheZuege :: Position -> [Position]
```

- Zugbaum files/Trees.hs

```
tttBaum = mkBaum moeglicheZuege startPosition
```

Bewertungsfunktion

- Berechnet für jede Position einen Spielwert, z.B.:
    - 1 -> Spieler 1 hat gewonnen
    - -1 -> Spieler 2 hat gewonnen
    - 0 -> unentschieden
- Abstrakte Bewertungsfunktion files/Trees.hs

```
wert :: Position -> Int
wert p = Prüfe diagonal,horizontal,vertikal
```

Abbildung zwischen Bäumen

- Motivation: Bestehende Baumstruktur um Bewertungsfunktion erweitern
- Baum als Implementierung der Klasse *Functor*
- konkret muss die Funktion *fmap* implementiert werden
- `fmap :: (Functor f) => (a -> b) -> f a -> f b`

- Implementierung von *fmap* files/Trees.hs

```
instance Functor Baum where
        fmap f (Knoten x c) = Knoten (f x) (map (fmap f) c)
```

- fmap ist strukturerhaltend
- Konstruktion des Wertebaums files/Trees.hs

```
werteBaum = fmap wert tttBaum
```

- abgebildet: Baum Position -> Baum Int

Auswertungsstrategien

- Motivation: Bestehende, bewertete Baumstruktur auswerten (welcher Knoten liefert maximales Ergebnis?)
- MinMax-Strategie files/Trees.hs

```
maximiere :: (Ord n) => (Baum n) -> n
maximiere (Knoten x []) = x
maximiere (Knoten x c) = maximum $ map minimiere c
```

- MinMax-Strategie files/Trees.hs

```
minimiere :: (Ord n) => (Baum n) -> n
```

```haskell
minimiere (Knoten x []) = x
minimiere (Knoten x c) = minimum $ map maximiere c
```

- Beipiel: Stellung bewerten, was kann maximal erreicht werden?:
- Spielwert errechnen files/Trees.hs

  ```haskell
  spielwert = maximiere werteBaum
  ```

- oder ausgeschrieben:
- `spielwert = maximiere $ fmap wert $ mkBaum moeglicheZuege startPosition`

Fazit

- Bei komplexeren Spielen müssen Bäume abgeschnitten werden ...
- Aber: Bäume können trotzdem unendlich definiert und später abgeschnitten werden

Modularisierung

- Algorithmen können auch für andere Spiele angewandt werden (z.B. Schach), denn
- die Baumdefinition bleibt gleich.
- Nur Zug- und Bewertungsfunktion müssen angepasst werden
- + ggf. Optimierung

Streams

Streams als unendliche Listen

- Unendliche Listen können als Datenströme genutzt werden.
- Einfache interaktive Programme können mit der Funktion interact realisiert werden.

Signatur von interact

```haskell
interact :: (String -> String) -> IO ()
```

- So lässt sich z.B. das shell-Kommando cat nachbauen:

Abgespeckte Version von cat files/mycat.hs

```haskell
main :: IO ()
main = interact id
```

Weitere shell-Kommandos

- Viele shell-Kommandos arbeiten Zeilenbasiert
- Wrapper für Zeilenbasiertes Arbeiten files/mysort.hs

  ```haskell
  linify :: ([String] -> [String]) -> (String -> String)
  linify f = (unlines . f . lines)
  ```

- einfaches Sortierprogramm files/mysort.hs

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

```haskell
main = interact (linify sort)
```

- einfache Version von head files/myhead.hs

```haskell
main = interact (linify (take 10))
```

- einfache Version von tail files/mytail.hs

```haskell
main = interact (linify (\ls -> drop (length ls - 10) ls))
```