Unendliche Listen und Bäume

Helga Karafiat, Steffen Rüther

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Lazy Evaluation

- Unendliche Strukturen sind möglich, weil Haskell Parameter erst und soweit auswertet, wie sie wirklich benötigt werden
  - call by need
    - Nur benötigte Auswertung
    - Keine doppelte Auswertung
  - erlaubt das Arbeiten mit undefinierten Werten und potentiell unendlich großen Datenmengen

- Nicht ohne Nachteile:
  - Erschwerung der Analyse der Komplexität und der Korrektheit von Programmen
  - Ablauf nicht immer intuitiv

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Konstruktion von unendlichen Strukturen

- Liste mit einem sich immer wiederholenden Wert:

```
ones = 1 : ones
```

- Aufsteigende Listen

```
fromStep :: Int -> Int -> [Int]
from n m = n : from (n+m) m

naturals = [1 ..]
evens = [2,4 ..]
odds = [1,3 ..]
```

- Unendliche Folgen als Listen

```
squares = [x^2 | x <- naturals]
```

Konstruktion von unendlichen Strukturen

- Generierungs-Funktion

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

- Unendliche Bäume

```
data Tree a   = Leaf | Node a (Tree a) (Tree a)

t = Node 1 t t
```

Stolperfalle

Beispiel:

Suche alle Quadrate von x, die kleiner sind als 100.

- Generierung mit Hilfe von List Comprehension

```
quad100 = [x | x <- squares, x < 100]
```

entspricht:

```
quad100 = filter (<100) squares
```

- Lösung:

```
quad100' = takeWhile (<100) squares
```

Mögliche gültige Anweisungen für unendliche Listen…

map

dropWhile

zip

take

foldr

[m..] !! n

zipWith

takeWhile

concat

drop

head

span

filter

Lieber nicht auf unendlichen Listen verwenden…

foldl

reverse

length

sort

partition

…

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Das Sieb des Erathostenes

- Algoritmus zur Ermittlung der Primzahlen

1. Schreibe alle natürlichen Zahlen von 2 bis zu einer beliebigen Zahl n auf.
2. Streiche alle Vielfachen von 2 heraus.
3. Gehe zur nächstgrößeren nicht gestrichenen Zahl und streiche deren Vielfache heraus.
4. Wiederhole 3. sooft es geht.
5. Die übriggebliebenen Zahlen sind Primzahlen.

- Funktion primes

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

Das Sieb des Erathostenes

- Auswertung der ersten Schritte von Primes

```
primes
sieve [2..]
2 : sieve : [y|y <- [3..], y `mod` 2 > 0]
2 : sieve (3 : [y|y <- [4..], y `mod` 2 > 0]
2 : 3 : sieve [z|z <- [y|y <- [4..], y `mod` 2 > 0]
                  z `mod` 3 > 0]
…
2 : 3 : sieve [z|z <- [5, 7, 9, …], z `mod` 3 > 0]
…
2 : 3 : sieve [5, 7, 11, …]
…
```

Random Numbers

- Generierung einer Sequenz von Zufallszahlen
- Haskell Programm kann keine „echten“ Random Numbers erzeugen

Generator Funktion:

```
nextRand :: Int -> Int
nextRand n = (mult * n + inc) `mod` modulus

randomSequence :: Int -> [Int]
randomSequence = iterate nextRand

seed        = 16232
mult        = 76367
inc         = 22131
modulus     = 65536

randomSequence seed
=> [16232,63371,34904,55707,60232,49579,…
```

Random Numbers

- Skalierung der Zufallszahlen
- *„Ein Liste von Zufallszahl mit den Zahlen 1 bis 6"*

Skalierungs-Funktion:

```
scaleSequence :: Int -> Int -> [Int] -> [Int]
scaleSequence s t = map scale
  where
  scale n  = n `div` denom + s
  ranger   = t - s + 1
  denom    = modulus `div` ranger
```

```
=> [2,6,4,6,6,5,1,6,6,4,…
```

Warum unendliche Listen?

Konzeptuell sehr elegant:

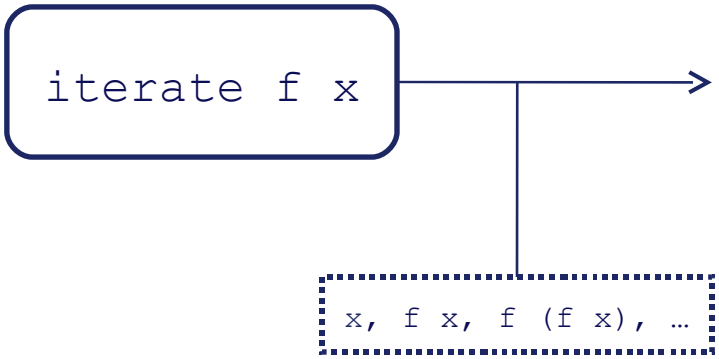
Man beschreibt eine unendliche Sequenz (Liste, Baum) und benutzt einen endlichen Teil davon.

Abstraktion

- Programme können besser abstrahiert und so einfacher implementiert werden
- Ein Argument wird nur ausgeführt, wenn es gebraucht wird
  - Beispiel: „*Finden der ersten n Primzahlen*"
- Man muss beim Programmieren nicht berücksichtigen wie groß die Sequenz werden könnte
  - Beispiel: *randomSequence*

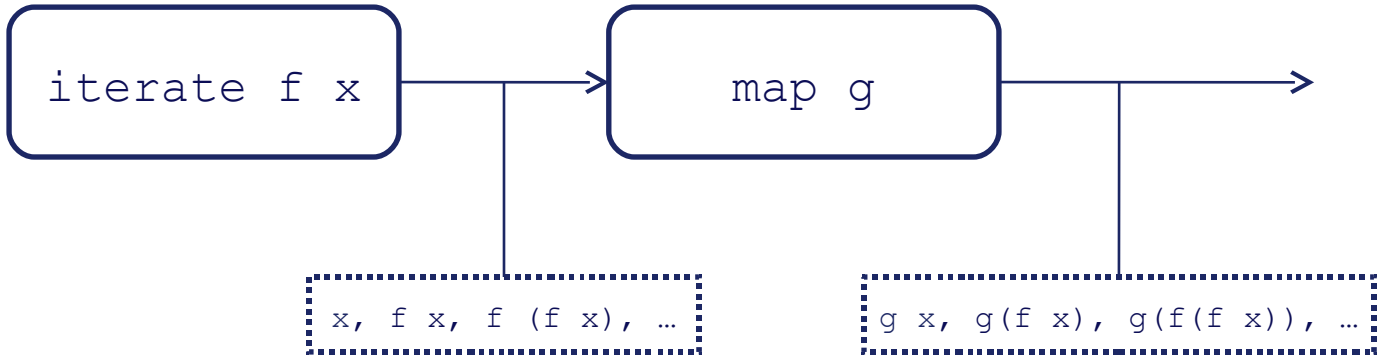
Warum unendliche Listen?

Generator



Transformer

Warum unendliche Listen?



Verknüpfung von Generator und Transformer

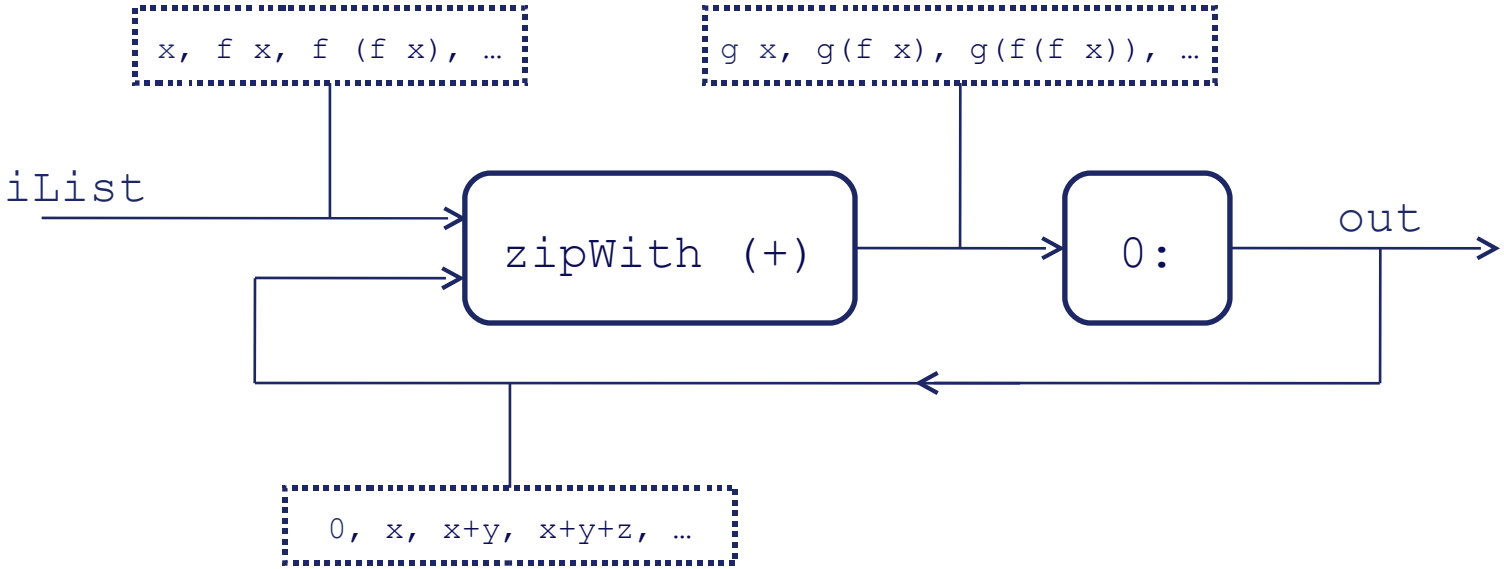
Modularisierung

- Trennung von Generierung und Transformation
- Beide Teile können unabhängig voneinander ausgetauscht werden
- Arbeiten und Programmieren auf rekursiven Strukturen

Warum unendliche Listen?

Beispiel „Fortlaufende Summe“:

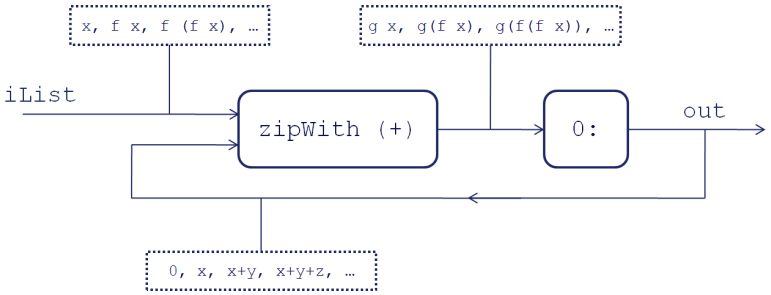
$[0, a_0, a_1, a_2, \ldots \rightarrow [0, a_0, a_0+a_1, a_0+a_1+a_2, \ldots$



Warum unendliche Listen?

Beispiel „Fortlaufende Summe“:

$[0, a_0, a_1, a_2, \ldots \rightarrow [0, a_0, a_0+a_1, a_0+a_1+a_2, \ldots$




```
listSums :: [Int] -> [Int]
listSums iList =
   out
   where
   out = 0 : zipWith (+) iList out

listSums [1..]
   => out
   => 0 : zipWith (+) [1..] out
   => 0 : zipWith (+) [1..] (0:…)
   => 0 : 1+0 : zipWith (+) [2..] (0+1:…)
   => 0 : 1 : 2+1 : zipWith (+) [3..] (2+1:…)
   => …
```

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Unendliche Listen als Grenzwerte

In der Mathematik werden oft unendliche Objekte als Grenzwerte für unendliche Sequenzen von Approximationen verwendet.

Beispiele:

3, 3.1, 3.14, 3.141, 3.1415, ...

1, 1.5,
1.4166666666666665,
1.4142156862745097,
1.4142135623746899,
1.414213562373095

Unendliche Listen als Grenzwerte

- Berechnung der Wurzel von 2

```
wurzel_2 =
    limes (iterate f 1)
    where
    f x = x / 2.0 + 1.0 / x
```

- Limes für den Abbruch

```
limes (a : b : x) =
    if a == b then a
    else limes (b : x)
```

- Berechnung der Wurzel von n

```
wurzel z =
    limes (iterate f 1)
    where
    f x = (x + z / x) / 2.0
```

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Zyklische Strukturen

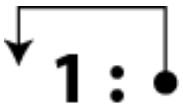
- Zurück zu ones:

```
ones = 1 : ones
```

- Auswertung von ones:

```
ones = 1 : ones
ones = 1 : 1 : ones
ones = 1 : 1 : 1 : ones
```

- Interne Darstellung in Haskell als Graph mit einer zyklischen Struktur:

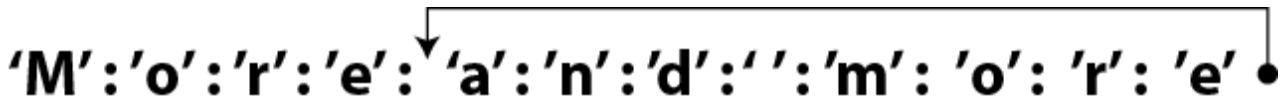


Zyklische Strukturen

- Weiteres Beispiel - more:

```
more :: String
more = "More" ++ andmore
       where andmore = "and more" ++
andmore
```

- Interne Darstellung in Haskell wiederrum als Graph mit einer zyklischen Struktur:



Zyklische Strukturen

- Vorteile von zyklischen Strukturen:
  - Konstanter Speicherplatzverbrauch
  - Effizienzvorteil (Laufzeitvorteil)

Zyklische Strukturen

- Realisierung von ones mit repeat
  - Definition von repeat

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

  - Definition von ones mit repeat

```
ones x = repeat 1
```

- Vorsicht! Keine zyklische Struktur mehr!
  - Auswertung der ersten 5 Elemente:

```
1 : 1 : 1 : 1 : 1 : repeat 1
```

  - nicht zyklisch, da bei jedem Auswertungschritt das Teilstück repeat 1 durch 1 : repeat 1 ersetzt wird.

- Definition von repeat für eine zyklische Struktur:

```
repeat x = xs where xs = x : xs
```

Zyklische Strukturen

- Iterate – Version 1

```
iterate1  :: (a -> a) -> a -> [a]
iterate1 f x = x : map f (iterate1 f x)
```

Erzeugt keine zyklische Struktur

- Iterate – Version 2

```
iterate2  :: (a -> a) -> a -> [a]
iterate2 f x = xs
               where xs = x : map f xs
```

Erzeugt eine zyklische Struktur

Zyklische Strukturen

- Iterate – Version 1

```
iterate f x = x : map f (iterate f x)
```

- Auswertungsschritte

```
iterate (2*) 1
=> 1 : map (2*) (iterate (2*) 1 )
=> 1 : 2 : map (2*) (map (2*) (iterate (2*) 1 ))
=> 1 : 2 : 4 : map (2*) (map (2*) (map (2*) (iterate (2*) 1
    )))
```

- Dauer der Auswertung der ersten Elemente n: $O(n^2)$

  -> sehr ineffizient

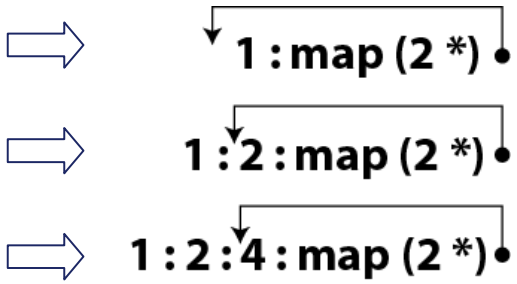
Zyklische Strukturen

- Iterate – Version 2

```
iterate f x = xs
            where xs = x : map f xs
```

- Interne Darstellung

```
iterate (2*) 1
```



- Dauer der Auswertung der ersten Elemente n: O(n)

  -> deutlich effizienter

Zyklische Strukturen – Das Hamming Problem

- Unendliche Zahlenliste mit folgenden Eigenschaften:
  - die Liste besitzt eine strikt aufsteigende Ordnung
  - das erste Element der Liste ist die 1
  - falls die Liste die Zahl x enthält, so enthält sie außerdem die Zahlen 2 * x, 3 * x und 5 * x
  - es sind keine weiteren Zahlen in der Liste enthalten

- Ersten Elemente der Liste:

  1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, ...

- Es existiert eine besonders effiziente Lösung, da sich die generierenden Funktionen monoton bezüglich (<) verhalten.

  Bsp: (2*): falls x < y gilt, gilt auch 2 x < 2 y

Zyklische Strukturen – Das Hamming Problem

- Lösung durch Funktion merge:
  erhält zwei unendliche Zahlenlisten in aufsteigender Ordnung als Argumente und überführt diese in eine unendliche Zahlenliste in aufsteigender Ordung ohne Duplikate.

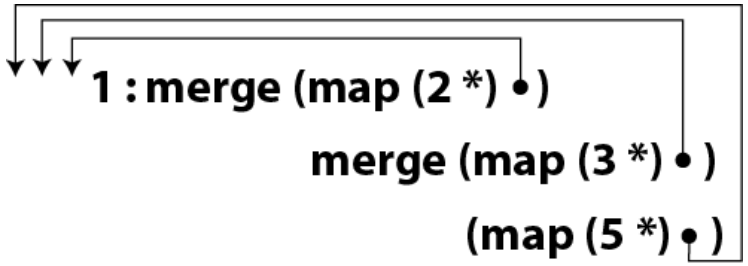
```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x : xs) (y : ys) | x < y  = x : merge xs (y : ys)
                        | x == y =   x : merge xs ys
                        | x > y  =   y : merge (x : xs) ys
```

- Deklaration von hamming mit Hilfe von merge

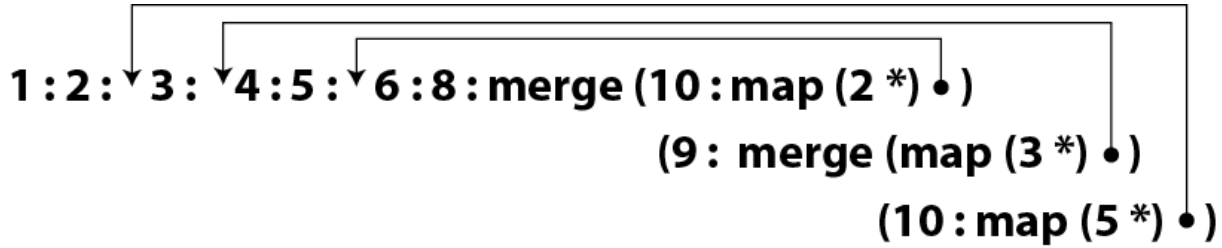
```
hamming :: [Integer]
hamming = 1 : merge (map (2*) hamming)
                    (merge (map (3*) hamming)
                           (map (5*) hamming))
```

Zyklische Strukturen – Das Hamming Problem

- Interne Darstellung der Funktion hamming



- Nach Auswertung der ersten sieben Schritte



- Dauer der Auswertung der ersten Elemente n: O(n)

Übersicht

- Grundlage: Lazy Evaluation
- Konstruktion von unendlichen Strukturen
- Verwendung von unendlichen Listen
- Unendliche Listen als Grenzwerte
- Zyklische Strukturen
- Dynamic Programming Example

Dynamic Programming Example

- Aufgabenstellung:

  Hunger! → …und das gewünschte Essen (Fingerfood) ist nur in 6er, 9er oder 20er Boxen erhältlich…

  Was machen wir nun, wenn wir genau 57 Stück essen wollen?

  Oder allgemeiner, kann man exakt n Stück kaufen?

- Randbedingungen
  - Nur 6er, 9er und 20er Boxen
  - Kaufbar sind Null bis mehrere dieser Boxen

Dynamic Programming Example

- Wenn man also i-6, i-9 oder i-20 Stück kaufen kann, kann man i Stück kaufen indem man die entsprechende Box dazukauft.

- Funktion *buyable*

```
import Data.Array

buyable n = r!n
    where r = listArray (0,n) (True : map f [1..n])
          f i = i >= 6 && r!(i-6) ||
                i >= 9 && r!(i-9) ||
                i >= 20 && r!(i-20)
```

Dynamic Programming Example

- Erweitung der Funktion *buyable* auf die Funktion *buy*
  - Es soll entweder ein Tripel mit der Anzahl der jeweils benötigten Boxen oder Nothing ausgegeben werden.

- Funktion *buy*

```
buy n = r!n
     where r = listArray (0,n) (Just (0,0,0) : map f [1..n])
           f i = case attempt (i-6)
                 of Just(x,y,z) -> Just(x+1,y,z)
                    _ -> case attempt (i-9)
                         of Just(x,y,z) -> Just(x,y+1,z)
                            _ -> case attempt (i-20)
                                 of Just(x,y,z) -> Just(x,y,z+1)
                                    _ -> Nothing
       attempt x = if x>=0 then r!x else Nothing
```

Dynamic Programming Example

- Funktion *buy* mit Hilfe von Monaden

```
buy' n = r!n
   where r = listArray (0,n) (Just (0,0,0) : map f [1..n])
         f i = do (x,y,z) <- attempt (i-6)
                  return (x+1,y,z)
                `mplus`
                do (x,y,z) <- attempt (i-9)
                   return (x,y+1,z)
                `mplus`
                do (x,y,z) <- attempt (i-20)
                   return (x,y,z+1)
         attempt x = guard (x>=0) >> r!x
```

Quellen

- **Bird, Richard**
  Introduction to Functional Programming using Haskell
  2nd Edition

- **Thompson, Simon**
  The Craft of Functional Programming
  2nd Edition

- http://www.haskell.org/haskellwiki/Dynamic_programming_example