Unendliche Listen

Funktionale Programmierung (Master)

FH Wedel

Daniel Jarosch & Karsten Thiele

18.12.2007

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Rückblick
- Grenzwerte
- Verhalten
- Zyklische Strukturen
- Beispiel: Stein-Schere-Papier
- Streambasierte Interaktionen

Inhalt

- **Rückblick**
- Grenzwerte
- Verhalten
- Zyklische Strukturen
- Beispiel: Stein-Schere-Papier
- Streambasierte Interaktionen

Rückblick

- Beschreibung
  - normal [n ..]
  - list comprehension [x | x <- [n ..]]
- Vergleich zu endlichen Listen
  - head [m ..] = m
  - take n [m ..] = [m .. (m+n)]
  - [m ..] !! n = m + n
  - …
- Programme arbeiten mit unendlichen Listen
- vollständige Induktion

Rückblick

- Bildung endlicher Liste mit list comprehensions?
  - [square x | x <- [0 ..] , square x < 10]
  - filter (< 10) (map square [0 ..])
  - Warum stoppt filter nicht nach dem Element 9?

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

Rückblick

- Bildung endlicher Liste mit list comprehensions?
  - [square x | x <- [0 ..] , square x < 10]
  - takeWhile (< 10) (map square [0 ..])
  - Ist diese Variante besser?

```
takeWhile :: (Integer -> Bool) -> [Integer] -> [Integer]
takeWhile p []           = []
takeWhile p (x : xs)     = if p x
                           then x : takeWhile p xs
                           else []
```

  - keine Endlosschleife

Inhalt

- Rückblick
- **Grenzwerte**
- Verhalten
- Zyklische Strukturen
- Beispiel: Stein-Schere-Papier
- Streambasierte Interaktionen

Grenzwerte

- unendliche Sequenzen werden approximiert
  - Annäherung an das Objekt
  - Beispiel: Pi = 3.1415926…
    - 3
    - 3.1
    - 3.14
    - 3.141
    - …

Grenzwerte

- unendliche Liste kann approximiert werden
  - Beispiel: [1 ..]
    - $\bot$
    - $1:\bot$
    - $1:2:\bot$
    - $1:2:3:\bot$
  - Untersequenz zum Beispiel
    - $\bot$
    - $1:2:\bot$
    - $1:2:3:4:\bot$
    - $1:2:3:4:5:6:\bot$

Grenzwerte

- Sonderfälle
  - nicht konvergierender Grenzwert
    - $\bot$
    - $1 : \bot$
    - $2 : 1 : \bot$
    - $3 : 2 : 1 : \bot$
  - endlicher Grenzwert
    - $\bot$
    - $1 : \bot$
    - $1 : 2 : \bot$
    - $1 : 2 : \bot$

Grenzwerte

- Ordnung über Approximationen
  - $x \subseteq y$
  - „x ist eine Approximation von y“
  - partielle Ordnung

  - Eigenschaften
    - reflexiv $x \subseteq x$
    - transitiv $x \subseteq y \wedge y \subseteq z \Rightarrow x \subseteq z$
    - anti-symmetrisch $x \subseteq y \wedge y \subseteq x \Rightarrow x = y$

- Ordnung über Approximationen
  - Zahlen, Booleans, Character und Aufzählungstypen
  - $x \subseteq y \equiv (x = \bot) \vee (x = y)$
    - x ist gleich y
    - x ist undefiniert
      - $\bot$ approximiert alles
      - $\bot$ ist ein Element der Ordnung
  - Ordnung über den Typen (a, b)
    - $\bot \subseteq (x, y)$
    - $(x, y) \subseteq (x', y') \equiv (x \subseteq x') \wedge (y \subseteq y')$
    - Beispiel: (Bool, Bool)
      - $\bot \subseteq (\bot, \bot) \subseteq (\bot, false) \subseteq (true, false)$

Grenzwerte

- Ordnung über Approximationen
  - Listen der Art [a]
    - $\bot \subseteq xs$
    - $[] \subseteq xs \equiv xs = []$
    - $(x : xs) \subseteq (y : ys) \equiv (x \subseteq y) \wedge (xs \subseteq ys)$
    - Beispiel:
      - $[1, \bot, 3] \subseteq [1,2,3]$
      - $1 : 2 : \bot \subseteq [1,2,3]$
      - $1 : 2 : \bot \not\subseteq [1, \bot, 3]$
    - …

Grenzwerte

- Ordnung über Approximationen
  - Listen der Art [a]
    - …
    - jede Approximationskette $x_1 \subseteq x_2 \subseteq ... \subseteq x_n$ besitzt einen Grenzwert, welcher ebenfalls vom Typ a ist
    - der Grenzwert erfüllt zwei Bedingungen
      - $x_n \subseteq \lim_{n\to\infty} x_n$ für alle n
      - wenn $x_n \subseteq y$, dann $\lim_{n\to\infty} x_n \subseteq y$ für alle n
    - gilt für jeden Typen
    - Approximationsketten mit dieser Eigenschaft werden mit „vollständig“ („complete“) beschrieben
    - …

- Ordnung über Approximationen
  - Listen der Art [a]
    - …
    - Wie kann eine Liste approximiert werden?
      - take
      - Ausnahme: approx 0 xs = ⊥

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

      - approx n [1] = [1] wenn n ≥ 2
      - $\lim_{n \to \infty}$ approx n xs = xs

- Berechenbare Funktionen
  - zwei Eigenschaften muss die Funktion f erfüllen
    - monoton
      - $x \subseteq y \Rightarrow f(x) \subseteq f(y)$
    - stetig
      - $f(\lim_{n\to\infty} x_n) \subseteq \lim_{n\to\infty} f(x_n)$
      - $x_1 \subseteq x_2 \subseteq \ldots \subseteq x_n$
  - Beispiel
    - gegeben sei die Approximation xs_n der Liste [1 ..]
    - auf die Approximation wird die Funktion map square xs_n angewandt
    - das Ergebnis ist identisch zur Approximation von map square [1 ..]

Grenzwerte

- Kettenvollständigkeit „chain completeness“
  - P als mathematische Aussage
  - P(xs) gilt für alle partiellen Listen von xs
  - Gilt P(xs) somit auch für unendliche Listen?
    - Grenzwert der Approximationskette ist $x_1 \subseteq x_2 \subseteq \ldots \subseteq x_n$ und es gilt $P(xs_i)$ für alle i
      - wenn P für approx gilt, gilt sie auch für unendliche Listen
      - Eigenschaft von P wird mit „chain complete“ bezeichnet

Inhalt

- Rückblick
- Grenzwerte
- **Verhalten**
- Zyklische Strukturen
- Beispiel: Stein-Schere-Papier
- Streambasierte Interaktionen

Verhalten

- nicht immer durch vollständige Induktion zu beweisen
- iterate f x = x : map f (iterate f x)
  - Definition von iterate
    - liefert eine unendliche Liste

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

  - Wie kann die Gültigkeit bewiesen werden?
    - kein adäquates Argument über das die Induktion laufen kann

Verhalten

- iterate f x = x : map f (iterate f x)
  - 1. Möglichkeit
    - zwei Listen sind gleich, wenn jedes Element gleich ist
    - xs !! n = ys !! n für alle natürlichen Zahlen n
    - Aussage ist falsch
      - xs = $\perp$ und ys = [$\perp$]
      - xs !! n = ys !! n = $\perp$
  - 2. Möglichkeit
    - Verwendung der Funktion approx
      - es gilt: $\lim_{n \to \infty}$ approx n xs = xs für alle Listen xs
      - daraus folgt: approx n xs = approx n ys für alle n, wenn xs = ys
    - Verallgemeinerung: Es kann gezeigt werden, dass gilt xs $\subseteq$ ys

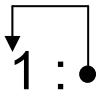
Inhalt

- Rückblick
- Grenzwerte
- Verhalten
- **Zyklische Strukturen**
- Beispiel: Stein-Schere-Papier
- Streambasierte Interaktionen

Zyklische Strukturen

- Rekursive Definition von Funktionen
  - 1. Beispiel: ones

```
ones :: [Int]
ones = 1 : ones
```

    - 1 : ones
    - 1 : 1 : ones
    - 1 : 1 : 1 : ones
    - ...
    - 

- Rekursive Definition von Funktionen
  - 2. Beispiel: more

```
more :: String
more = „More“ ++ andmore
       where andmore = „ and more“ ++ andmore
```

    - Ausgabe: putStr more

    - ‘M‘ : ‘o‘ : ‘r‘ : ‘e‘ : ‘ ‘ : ‘a‘ : ‘n‘ : ‘d‘ : ‘ ‘ :‘m‘ : ‘o‘ : ‘r‘ : ‘e‘ : •

Zyklische Strukturen

- repeat
  - Generierung von unendlichen Listen

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

  - Realisierung der Funktion ones

```
ones = repeat 1
```

keine zyklische Struktur

    - 1 : 1 : 1 : 1 : 1 : repeat 1
  - Definition als zyklische Struktur

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

Zyklische Strukturen

- iterate
  - iterate f x = x : map f (iterate f x)
    - 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)))
    - …

keine zyklische Struktur

- iterate
  - Definition als zyklische Struktur

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

  - iterate (2 *) 1
  - 1 : map (2 *) •
  - 1 : 2 : map (2 *) •
  - 1 : 2 : 4 : map (2 *) •
  - …

Zyklische Strukturen

- Hamming Problem
  - Erzeugung einer Liste mit folgenden Eigenschaften
    - strikt wachsende Ordnung
    - beginnt mit 1
    - wenn die Zahl x in der Liste, dann auch 2*x, 3*x und 5*x
    - keine anderen Zahlen
  - 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, ...
  - „closure problem“
    - Initialisierungselemente
    - Generator-Funktionen
  - effiziente Lösung, da Generatorfunktionen monoton

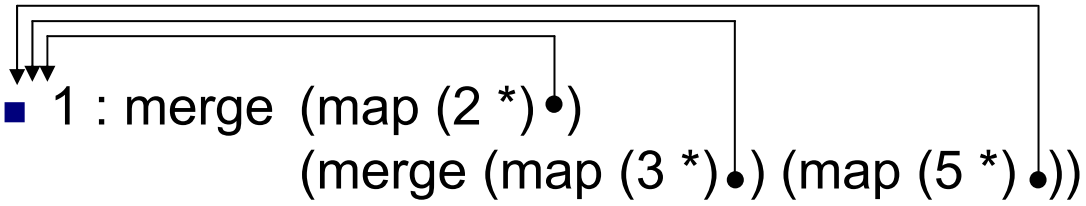
- Hamming Problem
  - Schlüssel zum Erfolg: merge

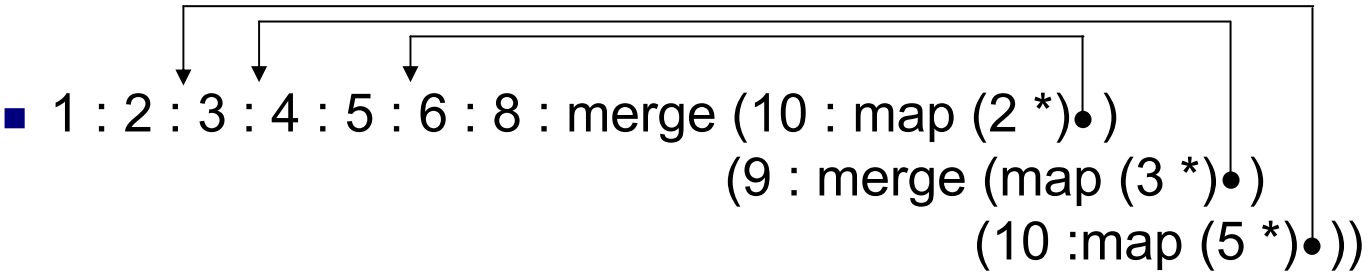
```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x : xs) (y : ys)
      | x < y   = x : merge xs (y : ys)
      | x == y  = x : merge xs ys
      | x > y   = y : merge (x :xs) ys
```

  - die hamming-Funktion

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

- Hamming Problem





- Hamming Problem
  - Verallgemeinerung
    - statt 2, 3 und 5 können beliebige positive Zahlen a, b, c verwendet werden
    - Definition

keine zyklische Struktur

```
hamming' :: Integer -> Integer -> Integer -> [Inte
hamming' a b c = 1 : merge(map(a *)hamming' a b c)
                         (merge(map(b *)hamming' a b c)
                                (map(c *)hamming' a b c))
```

Inhalt

- Rückblick
- Grenzwerte
- Verhalten
- Zyklische Strukturen
- **Beispiel: Stein-Schere-Papier**
- Streambasierte Interaktionen

Beispiel: Stein-Schere-Papier

- Nutzung unendlicher Listen, für Sequenz von Interaktionen zwischen Prozessen
- Datenstrukturen
  - grundlegende Typen

```
data Move    = Paper
             | Rock
             | Scissors

type Round   = (Move, Move)
```

Beispiel: Stein-Schere-Papier

- Spielzug
  - Vergabe der Punkte nach einer Runde

```
score :: Round -> (Int, Int)
score (x, y)
      | x `beats` y     = (1,0)
      | y `beats` x     = (0,1)
      | otherwise       = (0,0)
```

Beispiel: Stein-Schere-Papier

- Spielzug
  - Operator zur Bestimmung des Gewinners

```
beats :: Move -> Move -> Bool
x `beats` y = or [(m + 1 == n), (m == n + 2)]
              where
              m = code x
              n = code y
```

  - Umwandung von Aufzählungsdatentyp zu Int

```
code :: Move -> Int
code Paper        = 0
code Rock         = 1
code Scissors     = 2
```

Beispiel: Stein-Schere-Papier

- Spielstrategien
  - Typdefinition

```
type Strategy = [Move] -> Move
```

  - den letzten Zug vom Gegner „wiederholen“

```
recipro :: Strategy
recipro ms  = if null ms
              then Rock
              else last ms
```

Beispiel: Stein-Schere-Papier

- Spielstrategien
  - Spielzug erfolgt durch Analysierung des Gegners

```
smart :: Strategy
smart ms = if null ms
           then Rock
           else choose (count ms)
```

  - Anzahl der unterschiedlichen Züge

```
count :: [Move] -> (Int, Int, Int)
count = foldl (+++) (0, 0, 0)
```

Beispiel: Stein-Schere-Papier

- Spielstrategien
  - Funktion zum Berechnen der unterschiedlichen Züge

```
(+++) :: (Int, Int, Int) -> Move -> (Int, Int, Int)
(p, r, s) +++ Paper     = (p+1, r, s)
(p, r, s) +++ Rock      = (p, r+1, s)
(p, r, s) +++ Scissors  = (p, r, s+1)
```

  - Bestimmung vom nächsten Spielzug

```
choose :: (Int, Int, Int) -> Move
choose (p, r, s)
       | m < p     = Scissors
       | m < p + r = Paper
       | otherwise = Rock
       where m = random (p+r+s)
```

Beispiel: Stein-Schere-Papier

- Spielablauf
  - ein Spiel besteht aus mehreren Runden

```
rounds :: (Strategy, Strategy) -> [Round]
rounds (f, g) = (map last .
                 tail .
                 iterate (extend (f, g))) []
```

  - neuer Spielzug

```
extend :: (Strategy, Strategy) -> [Round] -> [Round]
extend (f, g) rs = rs ++ [(f (map snd rs),
                           g (map fst rs))]
```

Beispiel: Stein-Schere-Papier

- Punktevergabe
  - (End-) Ergebnis von n Spielen

```
match :: Int -> (Strategy, Strategy) -> (Int, Int)
match n = total . map score . take n . rounds
```

  - Berechnung der Gesamtpunktzahl

```
total :: [(Int, Int)] -> (Int, Int)
total = pair (sum . map fst, sum . map snd)
```

```
pair :: ((a -> b), (a -> c)) -> a -> (b, c)
pair (f, g) x = (f x, g x)
```

- Optimierung

```
type Strategy = [Move] -> [Move]
```

  - Neuimplementierung der Strategien

```
recipro :: Strategy
recipro ms = Rock : ms
```

```
smart :: Strategy
smart ms = Rock : map choose (counts ms)
```

```
counts :: [Move] -> [(Int, Int, Int)]
counts = tail . scanl (+++) (0, 0, 0)
```

Beispiel: Stein-Schere-Papier

- Optimierung
  - Redefinierung der Spielrunden

```
rounds :: (Strategy, Strategy) -> [Round]
rounds (f, g) = zip xs ys
                where
                xs = f ys
                ys = g xs
```

Beispiel: Stein-Schere-Papier

- Cheating
  - die Optimierung bietet keine Sicherheit gegen Schummeln

```
cheat :: Strategy
cheat xs     = map trumps xs
               where
               trumps :: Move -> Move
               trumps Paper    = Scissors
               trumps Rock     = Paper
               trumps Scissors = Rock
```

Beispiel: Stein-Schere-Papier

- Cheating
  - wie cheat: Aber bei bottom wird eine Liste geliefert

```
cunning :: Strategy
cunning xs = trumps (head xs) : cunning (tail xs)
```

  - erstes Spiel geschummelt

```
oneshot :: Strategy
oneshot xs = trumps (head xs) : recipro (tail xs)
```

  - die ersten zwei Spiele sind fair

```
devious :: Strategy
devious xs = take 2 (recipro xs) ++ cheat (drop 2 xs)
```

Beispiel: Stein-Schere-Papier

- Cheating
  - Wie kann das Schummeln unterbunden werden?
    - ein Spielzug muss ohne Informationen des gegnerischen Zuges berechnet werden können
    - alle Spielzüge müssen korrekt sein
  - Kann garantiert werden, dass nur ehrliche Strategien zugelassen werden?
    - mechanische Überprüfung nicht möglich
    - Funktion: police
      - zwingt die Strategie zur Ausgabe, bevor Input vorhanden ist
      - Schummelstrategien liefern Bottom

Beispiel: Stein-Schere-Papier

- Cheating
  - Deklarierung der „Schummel-Polizei“

```
police :: Strategy -> [Move] -> [Move]
police f xs = ys where ys = f (synch xs ys)
```

```
synch :: [Move] -> [Move] -> [Move]
synch (x : xs) (y : ys) = if (defined y)
                          then x : synch xs ys
                          else undefined
```

Beispiel: Stein-Schere-Papier

- Cheating
  - Überprüfung des Spielzuges

```
defined :: Move -> Bool
defined Paper    = True
defined Rock     = True
defined Scissors = True
defined x        = False
```

  - Anpassung der Rundenberechnung

```
rounds :: (Strategy, Strategy) -> [Round]
rounds (f, g) = zip xs ys
                where
                xs = police f ys
                ys = police g xs
```

Inhalt

- Rückblick
- Grenzwerte
- Verhalten
- Zyklische Strukturen
- Beispiel: Stein-Schere-Papier
- **Streambasierte Interaktionen**

Strombasierte Interaktion

- Interaktionen mit dem System immer nach dem gleichen Muster
  - Ausdruck → System → Berechnung → Ausgabe
- Oft auch andere Interaktionsformen nötig
  - Beispiel: Echos einer Eingabe auf Bildschirm

Strombasierte Interaktion

- Signatur eines strombasierten, interaktiven Programms

```
f :: String -> String
```

- Starten eines interaktiven Programms

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

- Beispiel: Kleinbuchstaben in Großbuchstaben konvertieren

```
map toUpper :: String -> String
    toUpper :: Char -> Char

   ?> interact(map toUpper)
```

- Abbruchkriterium einbauen

```
toUppers :: String -> String
toUppers = takeWhile (/= '.') . map toUpper
```

Vielen Dank für eure Aufmerksamkeit !!!