Unendliche Listen

Funktionale Programmierung in Haskell

Sebastian Schröder   Philip Chinery

Fachhochschule Wedel – University of Applied Sciences

11. Dezember 2009

Agenda

Agenda

1. Einleitung
   - Motivation
   - Aus der Mengenlehre
2. Eigenschaften
3. Ausprägungen
4. Beispiel
5. Unendliche Bäume
6. Zusammenfassung

Warum unendliche Listen?

Warum unendliche Listen?

- „Weil es geht“

Warum unendliche Listen?

- „Weil es geht“
- Unendliche Listen können in Haskell (partiell) verarbeitet werden
- Existieren auch außerhalb der Informatik
- Erleichtern die intuitive Definition mancher Funktionen

Zum Vortrag:

- Zwischenfragen sind ausdrücklich erwünscht
- Bei Fragen sind Antworten ausdrücklich erwünscht

Bekannte unendliche Listen

- bestimmte Zahlenmengen
  - natürliche Zahlen
  - alle geraden Zahlen
  - ...
- Ströme (Streams)
  - „stdin“
  - /dev/zero
  - ...

- unendliche Mengen: $\{x^2 | x \in \mathbb{N}; x^2 < 10\}$

- unendliche Mengen: $\{x^2 | x \in \mathbb{N}; x^2 < 10\}$
- Haskell: [square x | x <- [0..], square x < 10]

- unendliche Mengen: $\{x^2 | x \in \mathbb{N}; x^2 < 10\}$
- Haskell: [square x | x <- [0..], square x < 10]

Ergebnis

```
squares = [square x | x <- [0..], square x < 10]

> squares
> [0, 1, 4, 9
```

- unendliche Mengen: $\{x^2 | x \in \mathbb{N}; x^2 < 10\}$
- Haskell: [square x | x <- [0..], square x < 10]

Ergebnis

```
squares = [square x | x <- [0..], square x < 10]

> squares
> [0, 1, 4, 9
```

- Und dann?

Etwas Mathematik

- Der Computer ist nicht schlau genug
- Eigenschaften der erzeugenden Menge
  - Monotonität
  - Geeignete Abbruchbedingung

Etwas Mathematik

- Der Computer ist nicht schlau genug
- Eigenschaften der erzeugenden Menge
  - Monotonität
  - Geeignete Abbruchbedingung
- Was bietet sich an?

Etwas Mathematik

- Der Computer ist nicht schlau genug
- Eigenschaften der erzeugenden Menge
  - Monotonität
  - Geeignete Abbruchbedingung
- Was bietet sich an?
  - takeWhile

Quadrate < 10

```
squares' = [square x | x <- [0..]]
> takeWhile (<10) squares'
> [0, 1, 4, 9]
```

Unendliche Listen verarbeiten

- Grundsätzlich mit Haskell möglich
- Erfordert Bedarfsauswertung (lazy evaluation)
- => Funktionen dürfen nicht strikt sein

- Erfordert sorgfalt
  - Mathematische Eigenschaften ausnutzen
  - Vollständige Verarbeitung nicht möglich

Agenda

1. Einleitung
2. Eigenschaften
   - (Un)endliche Listen als Grenzwerte
   - Approximationsordnung
   - Beweis von Eigenschaften
3. Ausprägungen
4. Beispiel
5. Unendliche Bäume
6. Zusammenfassung

Mathematik: Unendliche Objekte

Mathematatik: Unendliche Objekte

- Grenzwerte von unendlichen Sequenzen von Approximationen

Mathematik: Unendliche Objekte

- Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Irrationale Zahl π (3,14159265358979...)

Mathematik: Unendliche Objekte

- Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Irrationale Zahl π (3,14159265358979...)
- Grenzwert der unendlichen Sequenz von rationalen Approximationen:
  3,0, 3,1, 3,14, 3,141, 3,1415, 3,14159, ...

Mathematik: Unendliche Objekte

- Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Irrationale Zahl π (3,14159265358979...)
- Grenzwert der unendlichen Sequenz von rationalen Approximationen:
  3,0, 3,1, 3,14, 3,141, 3,1415, 3,14159, ...
- Jedes weitere Element der Sequenz ist eine bessere Approximation des Grenzwertes

Unendliche Listen

Unendliche Listen

- Ebenso Grenzwerte von unendlichen Sequenzen von Approximationen

Unendliche Listen

- Ebenso Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Liste [1 ..]

Unendliche Listen

- Ebenso Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Liste [1 ..]
- Grenzwert der unendlichen Sequenz von partiellen Listen:
  $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$

Unendliche Listen

- Ebenso Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Liste [1 ..]
- Grenzwert der unendlichen Sequenz von partiellen Listen:
  $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- $\perp$ (undefiniert) liefert keine Informationen über den Grenzwert

Unendliche Listen

- Ebenso Grenzwerte von unendlichen Sequenzen von Approximationen
- Beispiel: Liste [1 ..]
- Grenzwert der unendlichen Sequenz von partiellen Listen:
  $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- $\perp$ (undefiniert) liefert keine Informationen über den Grenzwert
- Durch hinzufügen eines definierten Wertes ist jedes zusätzliche Element der Sequenz eine bessere Approximation des Grenzwertes

Konvergenz von Sequenzen

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- Konvergiert gegen den Grenzwert [1 ..]

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- Konvergiert gegen den Grenzwert [1 ..]

- $\perp, 1 : 2 : \perp, 1 : 2 : 3 : 4 : \perp, 1 : 2 : 3 : 4 : 5 : 6 : \perp, \ldots$

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, ...$
- Konvergiert gegen den Grenzwert [1 ..]

- $\perp, 1 : 2 : \perp, 1 : 2 : 3 : 4 : \perp, 1 : 2 : 3 : 4 : 5 : 6 : \perp, ...$
- Ist eine Untersequenz der obigen Sequenz
- Konvergiert ebenfalls gegen den Grenzwert [1 ..]

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- Konvergiert gegen den Grenzwert [1 ..]

- $\perp, 1 : 2 : \perp, 1 : 2 : 3 : 4 : \perp, 1 : 2 : 3 : 4 : 5 : 6 : \perp, \ldots$
- Ist eine Untersequenz der obigen Sequenz
- Konvergiert ebenfalls gegen den Grenzwert [1 ..]

- $\perp, 1 : \perp, 2 : 1 : \perp, 3 : 2 : 1 : \perp, \ldots$

Konvergenz von Sequenzen

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : 3 : \perp, 1 : 2 : 3 : 4 : \perp, \ldots$
- Konvergiert gegen den Grenzwert [1 ..]

- $\perp, 1 : 2 : \perp, 1 : 2 : 3 : 4 : \perp, 1 : 2 : 3 : 4 : 5 : 6 : \perp, \ldots$
- Ist eine Untersequenz der obigen Sequenz
- Konvergiert ebenfalls gegen den Grenzwert [1 ..]

- $\perp, 1 : \perp, 2 : 1 : \perp, 3 : 2 : 1 : \perp, \ldots$
- Konvergiert gegen keinen Grenzwert

Endliche Grenzwerte

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein
- $\perp$, $1 : \perp$, [1], [1], ...

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein
- $\perp$, $1 : \perp$, [1], [1], ...
- Konvergiert gegen den Grenzwert [1]

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein
- $\perp$, 1 : $\perp$, [1], [1], ...
- Konvergiert gegen den Grenzwert [1]

- $\perp$, 1 : $\perp$, 1 : 2 : $\perp$, 1 : 2 : $\perp$, ...

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein
- $\perp$, $1 : \perp$, [1], [1], ...
- Konvergiert gegen den Grenzwert [1]

- $\perp$, $1 : \perp$, $1 : 2 : \perp$, $1 : 2 : \perp$, ...
- Konvergiert gegen den Grenzwert $1 : 2 : \perp$

Endliche Grenzwerte

- Grenzwerte von Sequenzen können endlich sein
- $\perp, 1 : \perp, [1], [1], \ldots$
- Konvergiert gegen den Grenzwert [1]

- $\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : \perp, \ldots$
- Konvergiert gegen den Grenzwert $1 : 2 : \perp$

- Endliche und partielle Listen sind Grenzwerte für Sequenzen, welche nur aus einer endlichen Anzahl unterschiedlicher Elemente bestehen

Approximationsordnung

- Formalisierung der Eigenschaft, dass unendliche Sequenzen von partiellen Listen gegen einen Grenzwert konvergieren

- Formalisierung der Eigenschaft, dass unendliche Sequenzen von partiellen Listen gegen einen Grenzwert konvergieren

Approximationsordnung

- $x \sqsubseteq y$ (x ist eine Approximation von y)

Approximationsordnung

- Formalisierung der Eigenschaft, dass unendliche Sequenzen von partiellen Listen gegen einen Grenzwert konvergieren

Approximationsordnung

- $x \sqsubseteq y$ (x ist eine Approximation von y)

- reflexiv ($x \sqsubseteq x$)
- transitiv ($x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$)
- antisymmetrisch ($x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$)

Approximationsordnung

- Formalisierung der Eigenschaft, dass unendliche Sequenzen von partiellen Listen gegen einen Grenzwert konvergieren

Approximationsordnung

- $x \sqsubseteq y$ (x ist eine Approximation von y)

- reflexiv ($x \sqsubseteq x$)
- transitiv ($x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$)
- antisymmetrisch ($x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$)

- partielle Ordnung

Definition Approximationsordnung für Zahlen, Buchstaben und alle weiteren Aufzählungstypen

- $x \sqsubseteq y \equiv (x = \bot) \vee (x = y)$
- $\bot$ (Bottom) approximiert jeden Wert
- flache Ordnung

Definition Approximationsordnung für Zahlen, Buchstaben und alle weiteren Aufzählungstypen

- $x \sqsubseteq y \equiv (x = \perp) \vee (x = y)$
- $\perp$ (Bottom) approximiert jeden Wert
- flache Ordnung

Definition Approximationsordnung für Tupel (α,β)

- $\perp \sqsubseteq (x,y)$
- $(x,y) \sqsubseteq (x',y') \equiv (x \sqsubseteq x') \wedge (y \sqsubseteq y')$
- keine flache Ordnung

Definition Approximationsordnung für Listen $[\alpha]$

- $\perp \sqsubseteq xs$
- $[\,] \sqsubseteq xs \equiv xs = [\,]$
- $(x : xs) \sqsubseteq (y : ys) \equiv (x \sqsubseteq y) \wedge (xs \sqsubseteq ys)$

Definition Approximationsordnung für Listen $[\alpha]$

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

Beispiele:

- $[1, \bot, 3] \sqsubseteq [1, 2, 3]$

Definition Approximationsordnung für Listen $[\alpha]$

- $\perp \sqsubseteq \mathit{xs}$
- $[\,] \sqsubseteq \mathit{xs} \equiv \mathit{xs} = [\,]$
- $(x : \mathit{xs}) \sqsubseteq (y : \mathit{ys}) \equiv (x \sqsubseteq y) \wedge (\mathit{xs} \sqsubseteq \mathit{ys})$

Beispiele:

- $[1, \perp, 3] \sqsubseteq [1, 2, 3]$
- $[1, 2, \perp] \sqsubseteq [1, 2, 3]$

Definition Approximationsordnung für Listen $[\alpha]$

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

Beispiele:

- $[1, \bot, 3] \sqsubseteq [1, 2, 3]$
- $[1, 2, \bot] \sqsubseteq [1, 2, 3]$

- Jedoch **nicht**:
  $[1, 2, \bot] \sqsubseteq [1, \bot, 3]$

Weitere Eigenschaft von Approximationsordnungen für jeden Typen α

Jede Kette von Approximationen $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ hat einen Grenzwert welcher ebenfalls zu α gehört.

Weitere Eigenschaft von Approximationsordnungen für jeden Typen α

Jede Kette von Approximationen $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ hat einen Grenzwert welcher ebenfalls zu α gehört.

Eigenschaften des Grenzwertes $lim_{n\to\infty}x_n$

- $\forall n\ x_n \sqsubseteq lim_{n\to\infty}x_n$
  Der Grenzwert ist eine obere Grenze für die Sequenz von Approximationen

Weitere Eigenschaft von Approximationsordnungen für jeden Typen α

Jede Kette von Approximationen $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ hat einen Grenzwert welcher ebenfalls zu α gehört.

Eigenschaften des Grenzwertes $lim_{n\to\infty} x_n$

- $\forall n \; x_n \sqsubseteq lim_{n\to\infty} x_n$
  Der Grenzwert ist eine obere Grenze für die Sequenz von Approximationen
- Falls $x_n \sqsubseteq y$ für alle n gilt, dann ist $lim_{n\to\infty} x_n \sqsubseteq y$.
  Der Grenzwert ist die kleinste obere Grenze.

Haskell Funktion zur Approximationsberechnung

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

Haskell Funktion zur Approximationsberechnung

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

> approx 0 [1]
> undefined

> approx 1 [1]
> 1 : undefined

> approx 2 [1]
> 1 : [ ]
```

Haskell Funktion zur Approximationsberechnung

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

> approx 0 [1]
> undefined

> approx 1 [1]
> 1 : undefined

> approx 2 [1]
> 1 : [ ]
```

Wichtige Eigenschaft

- $lim_{n \to \infty}$ approx n xs = xs
  Für alle Listen xs (endlich, partiell oder unendlich)

Kettenvollständig (Chain Complete)

- Angenommen eine Eigenschaft P(xs) gilt für alle partiellen Listen xs
- Der Grenzwert einer Kette von Approximationen $xs_0, xs_1, ...$ (partielle Listen) ist eine unendliche Liste

- Angenommen eine Eigenschaft P(xs) gilt für alle partiellen Listen xs
- Der Grenzwert einer Kette von Approximationen $xs_0, xs_1, \ldots$ (partielle Listen) ist eine unendliche Liste

Kettenvollständig (Chain Complete)

- P ist eine mathematische Zusicherung
- Wenn P(xs) für alle Approximationen gilt, dann gilt es auch für den Grenzwert
- Wenn jedoch $\neg P$ für alle Approximationen gilt, dann muss es nicht auch für den Grenzwert gelten

Beispiele

- $e_1 \sqsubseteq e_2$, wobei e_1, e_2 Ausdrücke auf xs sind
- Variablen in den Ausdrücken sind allquantifiziert

Beispiele: Kettenvollständig (Chain Complete)

Beispiele

- $e_1 \sqsubseteq e_2$, wobei e_1, e_2 Ausdrücke auf xs sind
- Variablen in den Ausdrücken sind allquantifiziert
- kettenvollständig

Beispiele

- $e_1 \sqsubseteq e_2$, wobei e_1, e_2 Ausdrücke auf xs sind
- Variablen in den Ausdrücken sind allquantifiziert
- kettenvollständig

- $\exists n\ drop\ n\ xs = \bot$

Beispiele

- $e_1 \sqsubseteq e_2$, wobei e_1, e_2 Ausdrücke auf xs sind
- Variablen in den Ausdrücken sind allquantifiziert
- kettenvollständig

- $\exists n\ drop\ n\ xs = \bot$
- **Nicht** kettenvollständig

Haskell Funktion Iterate

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

Haskell Funktion Iterate

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

{- Folgende Gleichungen wird dabei erfüllt: -}
iterate f x = x : map f (iterate f x)
{- bzw. -}
iterate f (f x) = map f (iterate f x)
```

Haskell Funktion Iterate

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

{- Folgende Gleichungen wird dabei erfüllt: -}
iterate f x = x : map f (iterate f x)
{- bzw. -}
iterate f (f x) = map f (iterate f x)
```

Problem: Kein Argument vorhanden, über welchem die Induktion ausgeführt werden kann.

1. Ansatz

- Zwei Listen sind gleich, wenn alle Elemente an gleichen Positionen gleich sind
- xs !! n = ys !! n für alle natürlichen Zahlen n

1. Ansatz

- Zwei Listen sind gleich, wenn alle Elemente an gleichen Positionen gleich sind
- xs !! n = ys !! n für alle natürlichen Zahlen n
- Gegenbeispiel: xs = $\perp$ und ys = [$\perp$]

1. Ansatz

- Zwei Listen sind gleich, wenn alle Elemente an gleichen Positionen gleich sind
- xs !! n = ys !! n für alle natürlichen Zahlen n
- Gegenbeispiel: xs = $\perp$ und ys = [$\perp$]

2. Ansatz

- Nutzung der Funktion approx mit der Eigenschaft $lim_{n \to \infty}$ approx n xs = xs

Vollständige Induktion?

1. Ansatz

- Zwei Listen sind gleich, wenn alle Elemente an gleichen Positionen gleich sind
- xs !! n = ys !! n für alle natürlichen Zahlen n
- Gegenbeispiel: xs = $\perp$ und ys = [$\perp$]

2. Ansatz

- Nutzung der Funktion approx mit der Eigenschaft $lim_{n \to \infty}$ approx n xs = xs
- Somit $\forall n$ approx n xs = approx n ys $\to$ xs = ys
- Und $\forall n$ approx n xs $\sqsubseteq$ approx n ys $\to$ xs $\sqsubseteq$ ys

fhwedel UNIVERSITY OF APPLIED SCIENCES

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Linke Seite der Gleichung -}
approx (n+1) (iterate f (f x))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Linke Seite der Gleichung -}
approx (n+1) (iterate f (f x))

{- Definition iterate -}
approx (n+1) (f x : iterate f (f (f x)))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Linke Seite der Gleichung -}
approx (n+1) (iterate f (f x))

{- Definition iterate -}
approx (n+1) (f x : iterate f (f (f x)))

{- Definition approx -}
f x : approx n (iterate f (f (f x)))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Linke Seite der Gleichung -}
approx (n+1) (iterate f (f x))

{- Definition iterate -}
approx (n+1) (f x : iterate f (f (f x)))

{- Definition approx -}
f x : approx n (iterate f (f (f x)))

{- Induktionsannahme -}
f x : approx n (map f (iterate f (f x)))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Rechte Seite der Gleichung -}
approx (n+1) (map f (iterate f x))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Rechte Seite der Gleichung -}
approx (n+1) (map f (iterate f x))

{- Definition iterate -}
approx (n+1) (map f (x : iterate f (f x)))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Rechte Seite der Gleichung -}
approx (n+1) (map f (iterate f x))

{- Definition iterate -}
approx (n+1) (map f (x : iterate f (f x)))

{- Definition map -}
approx (n+1) (f x : map f (iterate f (f x)))
```

Vollständige Induktion über n

```
approx n (iterate f (f x)) =
approx n (map f (iterate f x))

{- Induktionsvoraussetzung (n=0) -}
approx 0 xs = undefined

{- Induktionsschritt (n = n+1) -}
{- Rechte Seite der Gleichung -}
approx (n+1) (map f (iterate f x))

{- Definition iterate -}
approx (n+1) (map f (x : iterate f (f x)))

{- Definition map -}
approx (n+1) (f x : map f (iterate f (f x)))

{- Definition approx -}
f x : approx n (map f (iterate f (f x)))
```

1. Einleitung
2. Eigenschaften
3. Ausprägungen
   - Unendliche Strukturen
   - Zyklische Strukturen
4. Beispiel
5. Unendliche Bäume
6. Zusammenfassung

Funktion Repeat

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

Funktion Repeat

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

Funktion Iterate

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

Funktion Repeat

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

Funktion Iterate

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

Probleme:

- Keine Möglichkeit der Darstellung als zyklischen Graphen im Auswerter

Funktion Repeat

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

Funktion Iterate

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

Probleme:

- Keine Möglichkeit der Darstellung als zyklischen Graphen im Auswerter
- Laufzeit kann durch unendliche Definition negativ beeinflusst werden

Beispiel für Darstellung im Auswerter

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

Beispiel für Darstellung im Auswerter

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

Beispiel für Laufzeitauswirkungen

```
iterate (2x) 1
> 1 : map (2x) (iterate (2x) 1)
> 1 : 2 : map (2x) (map (2x) (iterate (2x) 1))
> 1 : 2 : 4 : map (2x) (map (2x) (map (2x) (iterate
    (2x) 1)))
```

Beispiel für Darstellung im Auswerter

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

Beispiel für Laufzeitauswirkungen

```
iterate (2x) 1
> 1 : map (2x) (iterate (2x) 1)
> 1 : 2 : map (2x) (map (2x) (iterate (2x) 1))
> 1 : 2 : 4 : map (2x) (map (2x) (map (2x) (iterate
    (2x) 1)))
```

Laufzeit: $O(n^2)$

Datenstruktur ones

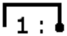
```
ones :: [Integer]
ones = 1 : ones
```

Zyklische Strukturen

Datenstruktur ones

```haskell
ones :: [Integer]
ones = 1 : ones
```

Darstellung im Auswerter



Funktion Iterate

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

Zyklische Strukturen

Funktion Iterate

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

Laufzeitauswirkungen

iterate (2x) 1

1 : map (2x)

1 : 2 : map (2x)

1 : 2 : 4 : map (2x)

Funktion Iterate

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

Laufzeitauswirkungen

iterate (2x) 1

$\boxed{1 : \text{map } (2x)}$

$1 : \boxed{2 : \text{map } (2x)}$

$1 : 2 : \boxed{4 : \text{map } (2x)}$

Laufzeit: $O(n)$

Agenda

Algorithmus

- Füge alle Zahlen größer 1 in das Sieb ein.
- Setze p auf die erste nicht gestrichene Zahl.
- Streiche alle Zahlen durch, die Vielfache von p sind.
- Setze p gleich der nächsten nicht durchgestrichenen Zahl.
- Weiter mit Schritt 2

Umsetzung in Haskell

- Füge alle Zahlen größer 1 in das Sieb ein.

Umsetzung in Haskell

- Füge alle Zahlen größer 1 in das Sieb ein.
- Und dann?

Umsetzung in Haskell

- Füge alle Zahlen größer 1 in das Sieb ein.
- Und dann?
- Direkte Übertragung nicht möglich
- Geeignete Implementierung nötig

Umsetzung in Haskell (2. Versuch)

- Nehme die Liste [2..]
- Erzeuge die Liste, die vielfache des ersten Elements nicht enthält
- Das erste Element ist eine Primzahl und wird in die Ergebnisliste eingefügt
- Rekursiver Aufruf mit der neuen Liste

Umsetzung in Haskell (2. Versuch)

- Nehme die Liste [2..]
- Erzeuge die Liste, die vielfache des ersten Elements nicht enthält
- Das erste Element ist eine Primzahl und wird in die Ergebnisliste eingefügt
- Rekursiver Aufruf mit der neuen Liste
- Geht das?

Umsetzung in Haskell (2. Versuch)

- Nehme die Liste [2..]
- Erzeuge die Liste, die vielfache des ersten Elements nicht enthält
- Das erste Element ist eine Primzahl und wird in die Ergebnisliste eingefügt
- Rekursiver Aufruf mit der neuen Liste
- Geht das?
- Ja, wenn man es geschickt macht

fhwedel UNIVERSITY OF APPLIED SCIENCES

Primzahlen

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

Analyse der Arbeitsschritte

- Ein Element der Liste abrufen (p) $O(n^2)$
  - Primzahllücke ist $O(n)$
  - Wegen Bedarfsauswertung muss die Berechnung auf jeder Rekusionsebene vorgenommen werden, also auf $O(n)$ Ebenen
- Aufstellen der Liste ohne Vielfache von p ($O(1)$)

Analyse der Arbeitsschritte

- Ein Element der Liste abrufen (p) $O(n^2)$
  - Primzahllücke ist $O(n)$
  - Wegen Bedarfsauswertung muss die Berechnung auf jeder Rekusionsebene vorgenommen werden, also auf $O(n)$ Ebenen
- Aufstellen der Liste ohne Vielfache von p ($O(1)$)
- Wir sind etwas großzügig; beweisbar:
- $O(n * (logn) * (loglogn))$

Analyse der Arbeitsschritte

- Ein Element der Liste abrufen (p) $O(n^2)$
  - Primzahllücke ist $O(n)$
  - Wegen Bedarfsauswertung muss die Berechnung auf jeder Rekusionsebene vorgenommen werden, also auf $O(n)$ Ebenen
- Aufstellen der Liste ohne Vielfache von p ($O(1)$)
- Wir sind etwas großzügig; beweisbar:
- $O(n * (logn) * (loglogn))$ | zu Hause ;-)

Erkenntnisse

- Unendliche Listen lassen sich leicht definieren
- Verarbeitung kann effizient erfolgen
- Kann intuitiv formuliert werden
- Komplexitätsanalyse funktioniert anders bei Bedarfsauswertung

Agenda

Ausprägungen

- Bäume mit unendlicher Tiefe
  - Feste Anzahl Kinder pro Knoten
  - Maximale Anzahl Kinder pro Knoten
- Bäume mit unendlicher Breite
  - Unendlich viele Kinder pro Knoten
  - Endliche Tiefe
- Bäume mit unendlicher Breite und Tiefe
  - Unendlich viele Kinder pro Knoten
  - Unendliche Tiefe
- Verallgemeinerungen (Unechte Bäume)
  - Kreise
  - Schlingen
  - Mehrfachkanten

Verarbeitung

- Vollständige Verarbeitung nicht möglich
- Verarbeitungsstrategien für große Bäume können (weitgehend) übernommen werden
- => Terminieren ohne zusätzliche Abbruchbedingungen nicht zwangsläufig

Strategien

- Breitensuche (breadth-first)
  - Für unendliche tiefe Bäume
- Tiefensuche (depth-first)
  - Für unendliche breite Bäume
- Bäume mit unendlicher Breite und Tiefe
  - Erfordern mehr Vorsicht
  - Konzept der Breitensuche verfeinern
    - Schrittweise Breite und Tiefe ausweiten

Vorkommen

- Entscheidungsbäume
- Landkarte auf einer Kugel
- Präfixbaum über alle möglichen Sätze

Vorkommen

- Entscheidungsbäume
- Landkarte auf einer Kugel
- Präfixbaum über alle möglichen Sätze
  - Das vollständige vorhandene und zukünftige niedergeschriebene Wissen der Welt in einer Datenstruktur von wenigen Byte

Unendliche Listen

- Erfordern etwas Eingewöhnung
- Können intuitiv Probleme Abbilden
- Erfordern Bedarfsauswertung

Vielen Dank für die Aufmerksamkeit!

Und vielen Dank an alle Beteiligten (Christoph und Thomas, Prof. Hoffmann, Alex Treptow, Till Tantau) für die LaTeX-Vorlage