

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 vom 22.11.2016: Funktionen Höherer Ordnung II und Effizienzaspekte

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

16.02.27 2017-01-17

1 [34]



Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
 - ▶ Einführung
 - ▶ Funktionen und Datentypen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Funktionen höherer Ordnung II und Effizienzaspekte
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

PI3 WS 16/17

2 [34]



Heute

- ▶ Mehr über map und fold
- ▶ map und fold sind nicht nur für Listen
- ▶ Effizient funktional programmieren

PI3 WS 16/17

3 [34]



map als strukturerhaltende Abbildung

- ▶ Der Datentyp $()$ ist **terminal**: es gibt für jeden Datentyp α genau eine total Abbildung $\alpha \rightarrow ()$
- ▶ **Struktur** (Shape) eines Datentyps T ist $T ()$
 - ▶ Gegeben durch kanonische Funktion $\text{shape} :: T \alpha \rightarrow T ()$, die α durch $()$ ersetzt
 - ▶ Für Listen: $[()] \cong \text{Nat}$

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{shape} \circ \text{map } f = \text{shape}$$

PI3 WS 16/17

4 [34]

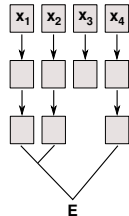


map und filter als Berechnungsmuster

- ▶ map, filter, fold als Berechnungsmuster:
 1. Anwenden einer Funktion auf **jedes** Element der Liste
 2. möglicherweise **Filtern** bestimmter Elemente
 3. **Kombination** der Ergebnisse zu Endergebnis E
- ▶ Gut parallelisierbar, skaliert daher als Berechnungsmuster für große Datenmengen (*big data*)
- ▶ Besondere Notation: Listenkompensation
 - ▶ $[f x \mid x \leftarrow \text{as}, g x] \equiv \text{map } f (\text{filter } g \text{ as})$
 - ▶ Beispiel:


```
digits str = [ord x - ord '0' \ x <- str, isDigit x]
```
 - ▶ Auch mit mehreren Generatoren:


```
idx = [ a: show i \ a <- ['a'.. 'z'], i <- [0.. 9]]
```



PI3 WS 16/17

5 [34]



foldr ist kanonisch

foldr ist die **kanonische strukturell rekursive Funktion**.

- ▶ Alle strukturell rekursiven Funktionen sind als Instanz von foldr darstellbar
- ▶ Insbesondere auch map und filter (siehe Übungsblatt)
- ▶ Es gilt: $\text{foldr } (:) [] = \text{id}$
- ▶ Jeder algebraischer Datentyp hat ein foldr

PI3 WS 16/17

6 [34]



fold für andere Datentypen

fold ist universell

Jeder algebraische Datentyp T hat genau ein foldr.

- ▶ Kanonische Signatur für T:
 - ▶ Pro Konstruktor C ein Funktionsargument f_C
 - ▶ Freie Typvariable β für T
- ▶ Kanonische Definition:
 - ▶ Pro Konstruktor C eine Gleichung
 - ▶ Gleichung wendet Funktionsparameter f_C auf Argumente an
- ▶ Anmerkung: Typklasse Foldable schränkt Signatur von foldr ein

data IL = Cons Int IL | Err String | Mt

```
foldIL :: (Int -> beta -> beta) -> (String -> beta) -> beta -> IL -> beta
foldIL f e a (Cons i il) = f i (foldIL f e a il)
foldIL f e a (Err str)  = e str
foldIL f e a Mt         = a
```

PI3 WS 16/17

7 [34]



fold für bekannte Datentypen

- ▶ Bool: Fallunterscheidung:

data Bool = False | True

```
foldBool :: beta -> beta -> Bool -> beta
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

- ▶ Maybe α : Auswertung

data Maybe α = Nothing | Just α

```
foldMaybe :: beta -> (alpha -> beta) -> Maybe alpha -> beta
foldMaybe b f Nothing = b
foldMaybe b f (Just a) = f a
```

- ▶ Als maybe vordefiniert

PI3 WS 16/17

8 [34]



fold für bekannte Datentypen

- ▶ Tupel: die uncurry-Funktion

```
foldPair :: (α → β → γ) → (α, β) → γ
foldPair f (a, b) = f a b
```

- ▶ Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat
foldNat :: β → (β → β) → Nat → β
foldNat e f Zero = e
foldNat e f (Succ n) = f (foldNat e f n)
```



fold für binäre Bäume

- ▶ Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

- ▶ Label **nur** in den Knoten

- ▶ Instanz von fold:

```
foldT :: (α → β → β → β) → β → Tree α → β
foldT f e Mt = e
foldT f e (Node a l r) = f a (foldT f e l) (foldT f e r)
```

- ▶ Instanz von map, kein (offensichtliches) Filter



Funktionen mit foldT und mapT

- ▶ Höhe des Baumes berechnen:

```
height :: Tree α → Int
height = foldT (λ_ l r → 1 + max l r) 0
```

- ▶ Inorder-Traversierung der Knoten:

```
inorder :: Tree α → [α]
inorder = foldT (λa l r → l ++ [a] ++ r) []
```



Kanonische Eigenschaften von foldT und mapT

- ▶ Auch hier gilt:

```
foldT Node Mt = id
mapT id = id
mapT f ∘ mapT g = mapT (f ∘ g)
shape (mapT f xs) = shape xs
```

- ▶ Mit shape :: Tree α → Tree ()



Das Labyrinth

- ▶ Das Labyrinth als variadischer Baum:

```
data VTree α = Node α [VTree α]
```

```
type Lab α = VTree α
```

- ▶ Auch hierfür foldT und mapT:

```
foldT :: (α → [β] → β) → VTree α → β
foldT f (Node a ns) = f a (map (foldT f) ns)
```

```
mapT :: (α → β) → VTree α → VTree β
mapT f (Node a ns) = Node (f a) (map (mapT f) ns)
```



Suche im Labyrinth

- ▶ Tiefensuche via foldT

```
dfts' :: Lab α → [Path α]
dfts' = foldT add where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

- ▶ Problem:

- ▶ foldT terminiert **nicht** für **zyklische** Strukturen
- ▶ Auch nicht, wenn add prüft ob a schon enthalten ist
- ▶ Pfade werden vom **Ende** konstruiert



Effizienzaspekte

- ▶ Zur Verbesserung der Effizienz:

- ▶ Analyse der **Auswertungsstrategie**
- ▶ ... und des **Speichermanagement**

- ▶ Der ewige Konflikt: **Geschwindigkeit vs. Platz**

- ▶ Effizienzverbesserungen durch

- ▶ **Endrekursion**: Iteration in funktionalen Sprachen
- ▶ **Striktheit**: **Speicherlecks** vermeiden (bei verzögerter Auswertung)

- ▶ Vorteil: Effizienz **muss nicht** im Vordergrund stehen



Endrekursion

Definition (Endrekursion)

Eine Funktion ist **endrekursiv**, wenn

- (i) es genau **einen** rekursiven Aufruf gibt,
- (ii) der **nicht** innerhalb eines **geschachtelten** Ausdrucks steht.

- ▶ D.h. darüber **nur Fallunterscheidungen**: **case** oder **if**
- ▶ Entspricht **goto** oder **while** in imperativen Sprachen.
- ▶ Wird in **Sprung** oder **Schleife** übersetzt.
- ▶ Braucht **keinen Platz** auf dem Stack.



Einfaches Beispiel

▶ In Haskell:

```
even x = if x >> 1 then even (x-2) else x == 0
```

▶ Übersetzt nach C:

```
int even (int x)
{ if (x>1) return (even (x-2))
  else return x == 0; }
```

▶ Äquivalente Formulierung:

```
int x; int even ()
{ if (x>1) { x -= 2; return even(); }
  return x == 0; }
```

▶ Iterative Variante mit Schleife:

```
int x; int even ()
{ while (x>1) { x -= 2; }
  return x == 0; }
```



Beispiel: Fakultät

▶ Fakultät **nicht** endrekursiv:

```
fac1 :: Integer -> Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

▶ Fakultät endrekursiv:

```
fac2 :: Integer -> Integer
fac2 n = fac' n 1 where
  fac' :: Integer -> Integer -> Integer
  fac' n acc = if n == 0 then acc
               else fac' (n-1) (n*acc)
```

- ▶ fac1 verbraucht Stack, fac2 nicht.
- ▶ Ist nicht merklich schneller?!



Beispiel: Listen umdrehen

▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] -> [a]
rev' [] = []
rev' (x:xs) = rev' xs ++ [x]
```

▶ Hängt auch noch hinten an — $O(n^2)$!

▶ Liste umdrehen, **endrekursiv** und $O(n)$:

```
rev :: [a] -> [a]
rev xs = rev0 xs [] where
  rev0 [] ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Beispiel: last (rev [1..10000])
- ▶ **Schneller** — warum?



Verzögerte Auswertung und Speicherlecks

- ▶ **Garbage collection** gibt **unbenutzten** Speicher wieder frei.
 - ▶ Unbenutzt: Bezeichner nicht mehr im **erreichbar**
- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird
 - ▶ Aber Achtung: **Speicherlecks**!
- ▶ Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
 - ▶ "Echte" Speicherlecks wie in C/C++ **nicht** möglich.
- ▶ Beispiel: fac2
 - ▶ Zwischenergebnisse werden **nicht** ausgewertet.
 - ▶ Insbesondere ärgerlich bei **nicht-terminierenden** Funktionen.



Striktheit

- ▶ **Strikte Argumente** erlauben Auswertung **vor** Aufruf
 - ▶ Dadurch **konstanter** Platz bei **Endrekursion**.
- ▶ **Erzwungene Striktheit**: seq :: $\alpha \rightarrow \beta \rightarrow \beta$
 - ▶ \perp 'seq' b = \perp
 - ▶ a 'seq' b = b
 - ▶ seq vordefiniert (nicht in Haskell definierbar)
 - ▶ (\$) :: $(a \rightarrow b) \rightarrow a \rightarrow b$ strikte Funktionsanwendung

```
f $! x = x 'seq' f x
```

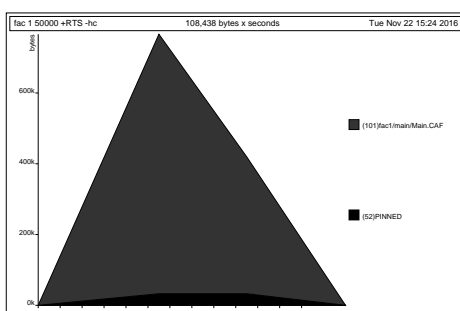
▶ ghc macht Striktheitsanalyse

▶ Fakultät in konstantem Platzaufwand

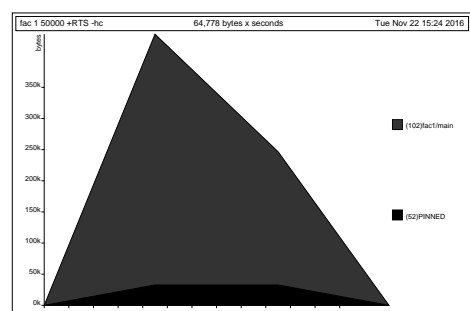
```
fac3 :: Integer -> Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc $ if n == 0 then acc
                    else fac' (n-1) (n*acc)
```



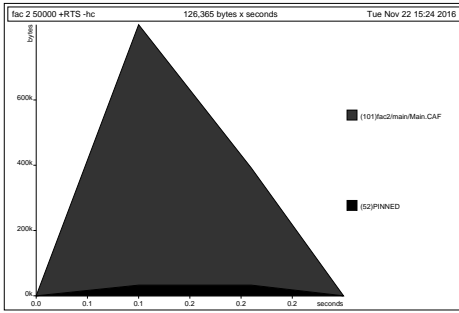
Speicherprofil: fac1 50000, nicht optimiert



Speicherprofil: fac1 50000, optimiert



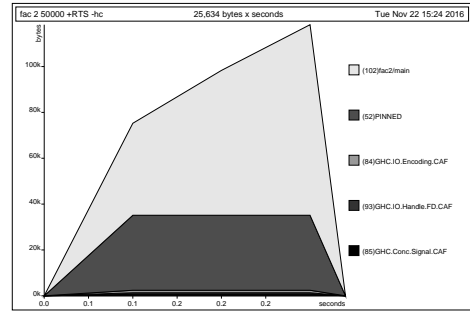
Speicherprofil: fac2 50000, nicht optimiert



PI3 WS 16/17

25 [34]

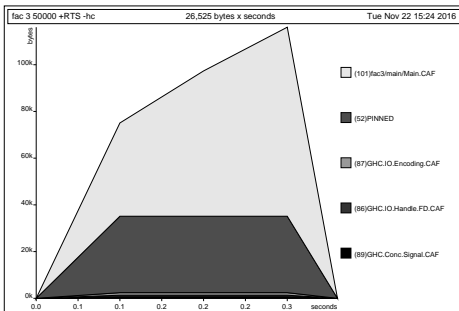
Speicherprofil: fac2 50000, optimiert



PI3 WS 16/17

26 [34]

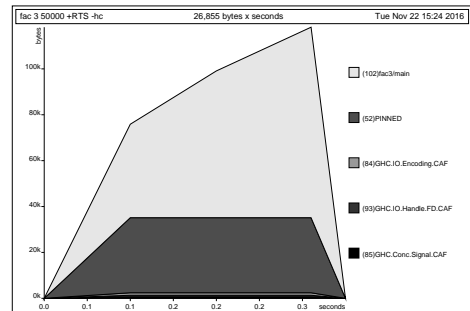
Speicherprofil: fac3 50000, nicht optimiert



PI3 WS 16/17

27 [34]

Speicherprofil: fac3 50000, optimiert



PI3 WS 16/17

28 [34]

Fazit Speicherprofile

- ▶ Endrekursion **nur** bei strikten Funktionen schneller
- ▶ Optimierung des *ghc*
 - ▶ Meist ausreichend für Striktheitsanalyse
 - ▶ Aber **nicht** für Endrekursion
- ▶ Deshalb:
 - ▶ Manuelle Überführung in Endrekursion sinnvoll
 - ▶ Compiler-Optimierung für Striktheit nutzen

PI3 WS 16/17

29 [34]

Überführung in Endrekursion

- ▶ Voraussetzung 1: Funktion ist **linear rekursiv**
- ▶ Gegeben Funktion

$$f' : S \rightarrow T$$

$$f' x = \text{if } B x \text{ then } H x \text{ else } (f' (K x)) \otimes (E x)$$
 - ▶ Mit $K : S \rightarrow S$, $\otimes : T \rightarrow T \rightarrow T$, $E : S \rightarrow T$, $H : S \rightarrow T$.
- ▶ Voraussetzung 2: \otimes assoziativ, $e : T$ neutrales Element
- ▶ Dann ist **endrekursive** Form:

$$f : S \rightarrow T$$

$$f x = g x e \text{ where}$$

$$g x y = \text{if } B x \text{ then } (H x) \otimes y \text{ else } g (K x) ((E x) \otimes y)$$

PI3 WS 16/17

30 [34]

Beispiel

- ▶ Länge einer Liste (nicht-endrekursiv)

```
length' :: [a] -> Int
length' xs = if null xs then 0
            else 1 + length' (tail xs)
```

- ▶ Zuordnung der Variablen:

$K(x) \mapsto \text{tail } x$	$B(x) \mapsto \text{null } x$
$E(x) \mapsto 1$	$H(x) \mapsto 0$
$x \otimes y \mapsto x + y$	$e \mapsto 0$

- ▶ Es gilt: $x \otimes e = x + 0 = x$ (0 neutrales Element)

PI3 WS 16/17

31 [34]

Beispiel

- ▶ Damit **endrekursive** Variante:

```
length :: [a] -> Int
length xs = len xs 0 where
  len xs y = if null xs then y -- was: y + 0
            else len (tail xs) (1 + y)
```

- ▶ Allgemeines **Muster**:

- ▶ Monoid (\otimes, e) : \otimes assoziativ, e neutrales Element.
- ▶ Zusätzlicher Parameter **akkumuliert** Resultat.

PI3 WS 16/17

32 [34]

Weiteres Beispiel: foldr vs. foldl

- ▶ foldr ist **nicht** endrekursiv:

```
foldr :: (α → β → β) → β → [α] → β
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- ▶ foldl ist **endrekursiv**:

```
foldl :: (α → β → α) → α → [β] → α
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- ▶ foldl' ist **strikt** und **endrekursiv**:

```
foldl' :: (α → β → α) → α → [β] → α
foldl' f a [] = a
foldl' f a (x:xs) = let a0 = f a x in a0 'seq' foldl' f a0 xs
```

- ▶ Für Monoid (\otimes, e) gilt: $\text{foldr } \otimes e l = \text{foldl } (\text{flip } \otimes) e l$



Zusammenfassung

- ▶ map und fold sind kanonische Funktionen höherer Ordnung, und für alle Datentypen definierbar
- ▶ map, filter, fold sind ein nützliches, skalierbares und allgemeines Berechnungsmuster
- ▶ Effizient funktional programmieren:
 - ▶ **Endrekursion**: while für Haskell
 - ▶ Mit Striktheit und Endrekursion **Speicherlecks** vermeiden.
 - ▶ Für Striktheit **Compileroptimierung** nutzen
- ▶ Nächste Woche: Funktionale Programmierung im Großen — Abstrakte Datentypen

