

Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 19.01.2011: Effizienzaspekte

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Inhalt

- ▶ Zeitbedarf: Endrekursion — **while** in Haskell
- ▶ Platzbedarf: Speicherlecks
- ▶ “Unendliche” Datenstrukturen
- ▶ Verschiedene andere Performancefallen:
 - ▶ Überladene Funktionen, Listen
- ▶ “Usual Disclaimers Apply”:
 - ▶ Erste Lösung: bessere Algorithmen
 - ▶ Zweite Lösung: Büchereien nutzen

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.

Beispiel: Fakultät

- ▶ **fac1 nicht** endrekursiv:

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

- ▶ **fac2** 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.

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])`

Überführung in Endrekursion

- ▶ Gegeben Funktion

$$f' : S \rightarrow T$$
$$f' x = \text{if } B x \text{ then } H x \text{ else } \phi (f' (K x)) (E x)$$

- ▶ Mit $K : S \rightarrow S$, $\phi : T \rightarrow T \rightarrow T$, $E : S \rightarrow T$, $H : S \rightarrow T$.

- ▶ Voraussetzung: ϕ assoziativ, $e : T$ neutrales Element

- ▶ Dann ist **endrekursive** Form:

$$f : S \rightarrow T$$
$$f x = g x e \text{ where}$$
$$g x y = \text{if } B x \text{ then } \phi (H x) y \text{ else } g (K x) (\phi (E x) y)$$

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) ↦ tail x      B(x) ↦ null x
E(x) ↦ 1           H(x) ↦ 0
φ(x,y) ↦ x+y       e ↦ 0
```

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

9 [33]

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

10 [33]

Endrekursive Aktionen

- ▶ **Nicht endrekursiv**:

```
getLines' :: IO String
getLines' = do str ← getLine
              if null str then return ""
              else do rest ← getLines'
                      return (str ++ rest)
```

- ▶ **Endrekursiv**:

```
getLines :: IO String
getLines = getit "" where
  getit res = do str ← getLine
                if null str then return res
                else getit (res ++ str)
```

11 [33]

Fortgeschrittene Endrekursion

- ▶ **Akkumulation** von Ergebniswerten durch **closures**

- ▶ closure: partiell applizierte Funktion

- ▶ Beispiel: die Klasse Show

- ▶ Nur Methode show wäre zu langsam ($O(n^2)$):

```
class Show a where
  show :: a → String
```

- ▶ Deshalb zusätzlich

```
showsPrec :: Int → a → String → String
show x = showsPrec 0 x ""
```

- ▶ String wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

12 [33]

Beispiel: Mengen als Listen

```
data Set a = Set [a]
```

Zu langsam wäre

```
instance Show a ⇒ Show (Set a) where
  show (Set elems) =
    "{" ++ intercalate ", " (map show elems) ++ "}"
```

Deshalb besser

```
instance Show a ⇒ Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems [] = ("{} " ++)
    showElems (x:xs) = ('{' :) ∘ shows x ∘ showl xs
    where showl [] = ('}' :)
          showl (x:xs) = (',' :) ∘ shows x ∘ showl xs
```

13 [33]

Effizienz durch "unendliche" Datenstrukturen

- ▶ Listen müssen nicht **endlich repräsentierbar** sein:

- ▶ Beispiel: "unendliche" Liste [2,2,2,...]

```
twos = 2 : twos
```

- ▶ Liste der natürlichen Zahlen:

```
nat = nats 0 where nats n = n : nats (n+1)
```

- ▶ Syntaktischer Zucker:

```
nat = [0 ..]
```

- ▶ Bildung von unendlichen Listen:

```
cycle :: [a] → [a]      repeat :: a → [a]
cycle xs = xs ++ cycle xs  repeat x = x : repeat x
```

- ▶ Nützlich für Listen mit unbekannter Länge
- ▶ **Obacht**: Induktion nur für **endliche** Listen gültig.

14 [33]

Berechnung der ersten n Primzahlen

- ▶ **Eratosthenes** — aber bis wo sieben?

- ▶ Lösung: Berechnung **aller** Primzahlen, davon die **ersten** n .

```
sieve :: [Integer] → [Integer]
sieve (p:ps) =
  p:(sieve (filter (\n → n `mod` p /= 0) ps))
```

- ▶ Keine Rekursionsverankerung (sieve [])

```
primes :: [Integer]
primes = sieve [2 ..]
```

- ▶ Von allen Primzahlen die **ersten** n :

```
firstprimes :: Int → [Integer]
firstprimes n = take n primes
```

15 [33]

Fibonacci-Zahlen

- ▶ Aus der Kaninchenzucht.

- ▶ Sollte jeder Informatiker kennen.

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

- ▶ Problem: **baumartige** Rekursion, **exponentieller** Aufwand.

16 [33]

Fibonacci-Zahlen als Strom

- ▶ Lösung: zuvor berechnete Teilergebnisse wiederverwenden.

- ▶ Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
fibs 1 1 2 3 5 8 13 21 34 55
tail fibs 1 2 3 5 8 13 21 34 55
tail (tail fibs) 2 3 5 8 13 21 34 55
```

- ▶ Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- ▶ n-te Fibonaccizahl mit `fibs !! n`
- ▶ Aufwand: **linear**, da `fibs` nur einmal ausgewertet wird.

17 [33]

Unendliche Datenstrukturen

- ▶ Endliche Repräsentierbarkeit für beliebige Datenstrukturen

- ▶ E.g. Bäume:

```
data Tree a = Null | Node (Tree a) a (Tree a)
            deriving Show

twoTree     = Node twoTree 2 twoTree
rightSpline n = Node Null n (rightSpline (n+1))
```

- ▶ `twoTree`, `twos` mit Zeigern darstellbar (e.g. Java, C)
- ▶ `rightSpline 0`, `nat` **nicht** mit darstellbar
- ▶ Damit beispielsweise auch **Graphen** modellierbar

18 [33]

Implementation und Repräsentation von Datenstrukturen

- ▶ Datenstrukturen werden intern durch **Objekte** in einem **Heap** repräsentiert

- ▶ Bezeichner werden an **Referenzen** in diesen Heap gebunden

- ▶ Unendliche Datenstrukturen haben zyklische Verweise

- ▶ Kopf wird nur einmal ausgewertet.

```
cycle (trace "Foo!" [5])
```

- ▶ **Anmerkung:** unendlich Datenstrukturen nur sinnvoll für **nicht-strikte** Funktionen

19 [33]

Speicherlecks

- ▶ **Garbage collection** gibt **unbenutzten** Speicher wieder frei.

- ▶ **Unbenutzt:** Bezeichner nicht mehr im **erreichbar**

- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird

- ▶ Aber Obacht: **Speicherlecks!**

- ▶ Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.

- ▶ "Echte" Speicherlecks wie in C/C++ **nicht** möglich.

- ▶ Beispiel: `getLines`, `fac2`

- ▶ Zwischenergebnisse werden **nicht** ausgewertet.

- ▶ Insbesondere ärgerlich bei **nicht-terminierenden** Funktionen.

20 [33]

Striktheit

- ▶ **Strikte Argumente** erlauben Auswertung **vor** Aufruf

- ▶ Dadurch **konstanter** Platz bei **Endrekursion**.

- ▶ **Erzwungene Striktheit:** `seq :: $\alpha \rightarrow \beta \rightarrow \beta$`

```
⊥ 'seq' b = ⊥
a 'seq' b = b
```

- ▶ `seq` vordefiniert (nicht in Haskell definierbar)

- ▶ `($!) :: (a → b) → a → b` strikte Funktionsanwendung

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

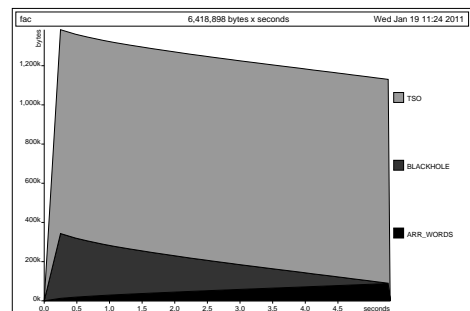
- ▶ `ghc` macht Striktheitsanalyse

- ▶ 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)
```

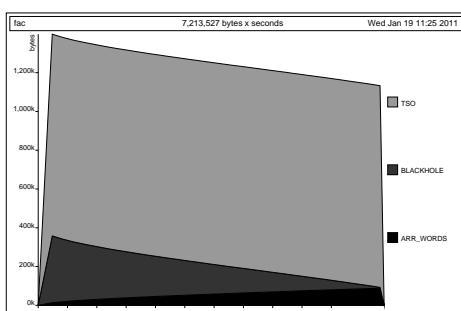
21 [33]

Speicherprofil: fac1 50000, nicht optimiert



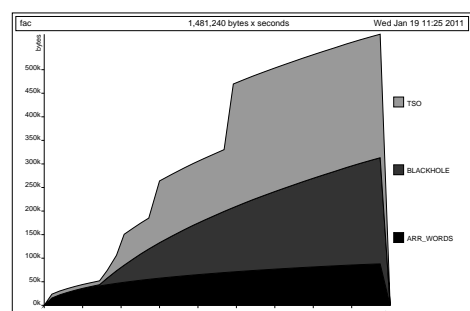
22 [33]

Speicherprofil: fac2 50000, nicht optimiert



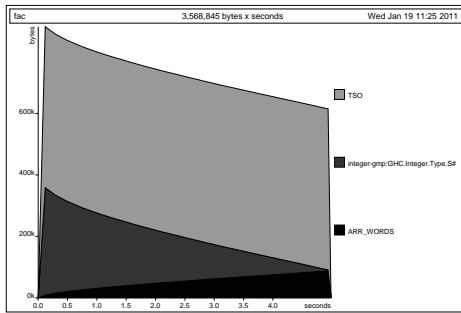
23 [33]

Speicherprofil: fac3 50000, nicht optimiert



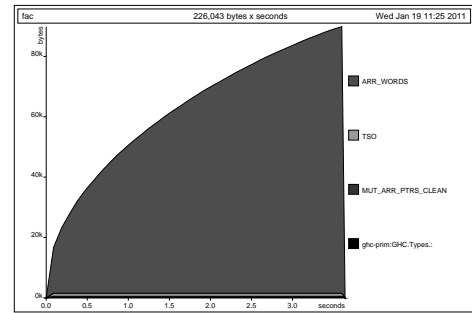
24 [33]

Speicherprofil: fac1 50000, optimiert



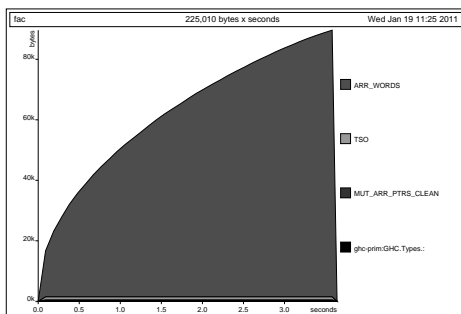
25 [33]

Speicherprofil: fac2 50000, optimiert



26 [33]

Speicherprofil: fac3 50000, optimiert



27 [33]

Fazit Speicherprofile

- ▶ Geschwindigkeitsgewinn durch Endrekursion **nur** mit Striktheit
- ▶ Optimierung des ghc meist ausreichend für Striktheitsanalyse, aber **nicht** für Endrekursion

28 [33]

foldr vs. foldl

- ▶ foldr ist **nicht** endrekursiv:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- ▶ foldl ist **endrekursiv**:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

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

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a [] = a
foldl' f a (x:xs) =
  let a' = f a x in a' `seq` foldl' f a' xs
```

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

29 [33]

Wann welches fold?

- ▶ foldl endrekursiv, aber traversiert immer die **ganze** Liste.

- ▶ foldl' ferner strikt und konstanter Platzaufwand

- ▶ Wann welches fold?

- ▶ Strikte Funktionen mit foldl' falten:

```
rev2 :: [a] -> [a]
rev2 = foldl' (flip (:)) []
```

- ▶ Wenn nicht die ganze Liste benötigt wird, mit foldr falten:

```
all :: (a -> Bool) -> [a] -> Bool
all p = foldr ((&&) o p) True
```

- ▶ Potenziell **unendliche** Listen **immer** mit foldr falten.

30 [33]

Überladene Funktionen sind langsam.

- ▶ Typklassen sind elegant aber **langsam**.

- ▶ Implementierung von Typklassen: **Verzeichnis** (dictionary) von Klassenfunktionen.

- ▶ Überladung wird zur **Laufzeit** aufgelöst

- ▶ Bei kritischen Funktionen: **Spezialisierung erzwingen** durch Angabe der Signatur

- ▶ NB: **Zahlen** (numerische Literale) sind in Haskell **überladen**!

- ▶ Bsp: facts hat den Typ Num a => a -> a

```
facts n = if n == 0 then 1 else n * facts (n-1)
```

31 [33]

Listen als Performance-Falle

- ▶ Listen sind **keine** Felder oder endliche Abbildungen

- ▶ Listen:

- ▶ Beliebig lang
- ▶ Zugriff auf n -tes Element in **linearer** Zeit.
- ▶ Abstrakt: frei erzeugter Datentyp aus Kopf und Rest

- ▶ Felder Array $ix \ a$ (Modul Array aus der Standardbücherei)

- ▶ Feste Größe (Untermenge von ix)
- ▶ Zugriff auf n -tes Element in **konstanter** Zeit.
- ▶ Abstrakt: Abbildung Index auf Daten

- ▶ Endliche Abbildung Map $k \ v$ (Modul Data.Map)

- ▶ Beliebige Größe
- ▶ Zugriff auf n -tes Element in **sublinearer** Zeit.
- ▶ Abstrakt: Abbildung Schlüsselbereich k auf Wertebereich v

32 [33]

Zusammenfassung

- ▶ **Endrekursion**: **while** für Haskell.
 - ▶ Überführung in Endrekursion meist möglich.
 - ▶ Noch besser sind strikte Funktionen.
- ▶ **Speicherlecks** vermeiden: Striktheit und Endrekursion
- ▶ Compileroptimierung nutzen
- ▶ Datenstrukturen müssen nicht endlich repräsentierbar sein
- ▶ Überladene Funktionen sind langsam.
- ▶ Listen sind keine Felder oder endliche Abbildungen.