Praktische Informatik 3: Funktionale Programmierung
Vorlesung 12 vom 15.01.2013: Effizienzaspekte

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Effizienzaspekte
  - Eine Einführung in Scala
  - Rückblich & 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
- Effizenzverbesserungen 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 x == 0 else even (x-2)
```

- Übersetzt nach C:

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

- Äquivalente Formulierung:

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

- Iterative Variante mit Schleife:

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


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

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

  - Mit $K : S \to S$, $\phi : T \to T \to T$, $E : S \to T$, $H : S \to T$.
- Voraussetzung: ϕ assoziativ, $e : T$ neutrales Element
- Dann ist endrekursive Form:

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

Beispiel

- Länge einer Liste (nicht-endrekursiv)

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

- Zuordnung der Variablen:

$K(x)$	$\mapsto$	tail x		$B(x)$	$\mapsto$	null x
$E(x)$	$\mapsto$	1		$H(x)$	$\mapsto$	0
$\phi(x,y)$	$\mapsto$	$x+y$		e	$\mapsto$	0

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

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.

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)
```


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 ($O(n)$).

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
```


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: getLines, fac2
  - Zwischenergebnisse werden nicht auswertet.
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Erzwungene Striktheit: seq :: $\alpha \to \beta \to \beta$

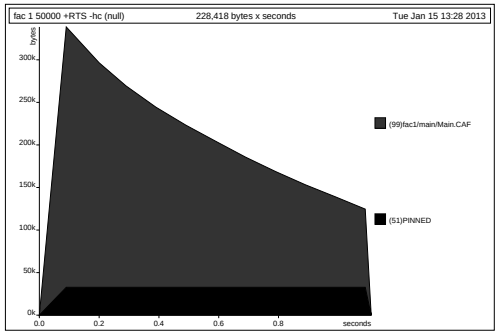
  $\bot$ `seq` b $=\ \bot$
  a `seq` b $=$ b
  - seq vordefiniert (nicht in Haskell definierbar)
  - ($!) :: (a-> b)-> a-> b strikte Funktionsanwendung

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

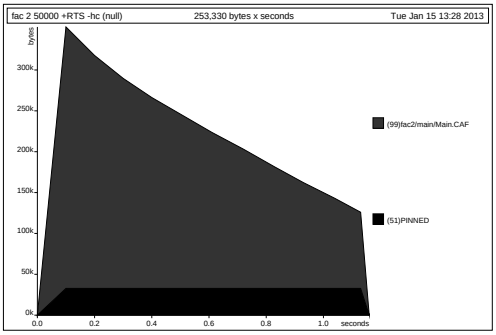
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```
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)
```

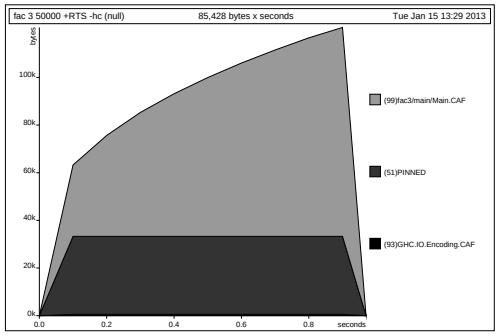
Speicherprofil: fac1 50000, nicht optimiert



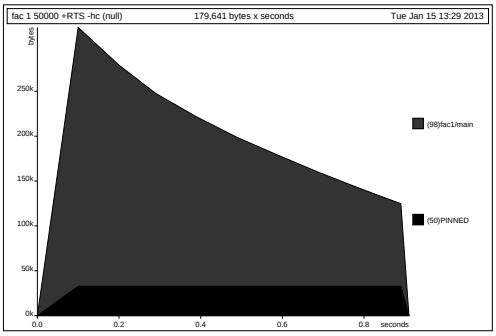
Speicherprofil: fac2 50000, nicht optimiert



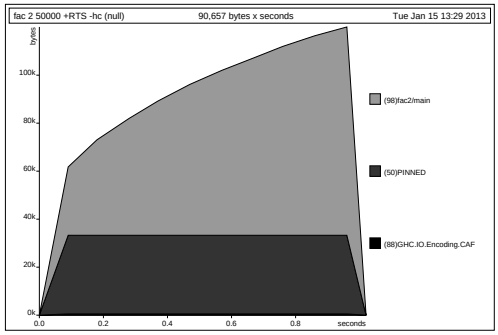
Speicherprofil: fac3 50000, nicht optimiert



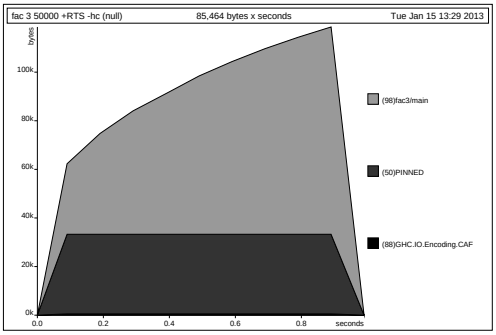
Speicherprofil: fac1 50000, optimiert



Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, optimiert



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

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: foldr ϕ e l = foldl (flip ϕ) e l

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 ((&&) ∘ p) True
```

  - Potenziell unendliche Listen immer mit foldr falten.

Effizienz durch "unendliche" Datenstrukturen

- Listen müssen nicht endlich repräsentierbar sein:
  - Nützlich für Listen mit unbekannter Länge
  - Obacht: Induktion nur für endliche Listen gültig.
- Beispiel: Fibonacci-Zahlen
  - Aus der Kaninchenzucht.
  - Sollte jeder Informatiker kennen.

```
fib' :: Int→ Integer
fib' 0 = 1
fib' 1 = 1
fib' n = fib' (n−1)+ fib' (n−2)
```

  - Problem: baumartige Rekursion, exponentieller Aufwand.

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.

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

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: `facs` hat den Typ `Num a=> a-> a`

```
facs n = if n == 0 then 1 else n* facs (n−1)
```


Listen als Performance-Falle

- Listen sind keine Felder oder endliche Abbildungen
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit ($O(n)$)
  - 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 ($O(1)$)
  - Abstrakt: Abbildung Index auf Daten
- Endliche Abbildung `Map k v` (Modul `Data.Map`)
  - Beliebige Größe
  - Zugriff auf n-tes Element in sublinearer Zeit ($O(\log n)$)
  - Abstrakt: Abbildung Schlüsselbereich `k` auf Wertebereich `v`
  - Sonderfall: `Set k` ≡ `Map k Bool`

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.