Effizienz

Wintersemester 2007/2008

Kim Kirchbach (Inf6310)
Mirco Schenkel (Inf6311)

Inhalt

- Lazy Evaluation
- Komplexität
- Parameterakkumulation
- Tupling
- Speicherplatz
- Fazit

Lazy Evaluation

Effizenz vs Effektivität

- Die **Effizienz** eines Algorithmus ist seine Sparsamkeit bezüglich der Ressourcen, Zeit und Speicherplatz, die er zur Lösung eines festgelegten Problems beansprucht
- **Effektivität** wird unterschiedlich definiert, je nachdem, welchen Sachverhalt man damit bezeichnet. Aus der allgemeinen wissenschaftlichen Perspektive bedeutet Effektivität das Verhältnis von erreichtem Ziel zu definiertem Ziel
- Dies ist im Unterschied zur Effizienz unabhängig vom zur Zielerreichung nötigen Aufwand. Effektiv arbeiten bedeutet, unter Einsatz aller Mittel ein Ziel zu erreichen, effizient arbeiten hingegen bedeutet, ein Ziel mit möglichst geringem Mitteleinsatz zu erreichen Effizienz setzt also Effektivität voraus und geht über diese noch hinaus
- Effektivität ist ein Maß für die Zielerreichung (Wirksamkeit, Output, Qualität der Zielerreichung)
- Effizienz ist ein Maß für die Wirtschaftlichkeit (Kosten-Nutzen-Relation)

Lazy Evaluation

Innerste Reduktion

```
{leftmost innermost reduction}
  square(3+4)
= {definition of +}
  square 7
= {definition of square}
  7 x 7
= {definition of x}
  49
```

Äußere Reduktion

```
{leftmost outermost reduction}
  square(3+4)
= {definition of square}
  (3+4) x (3+4)
= {definition of +}
  7 x (3+4)
= {definition of +}
  7 x 7
= {definition of x}
  49
```

Lazy Evaluation

Innerste Reduktion

```
{leftmost innermost reduction}
  fst(square 4, square 2)
= {definition of square}
  fst(4 x 4, square 2)
= {definition of x}
  fst(16, square 2)
= {definition of square}
  fst(16, 2 x 2)
= {definition of x}
  fst(16,4)
= {definition of fst}
  16
```

Äußere Reduktion

```
{leftmost outermost reduction}
  fst(square 4, square 2)
= {definition of fst}
  square 4
= {definition of square}
  4 x 4
= {definition of x}
 16
```

Lazy Evaluation

Innerste Reduktion

```
{leftmost innermost reduction}
  fst(square 4, undefined)
= {definition of square}
  fst(4 x 4, undefined)
= {definition of x}
  fst(16, undefined)
= {ERROR}
```

Äußere Reduktion

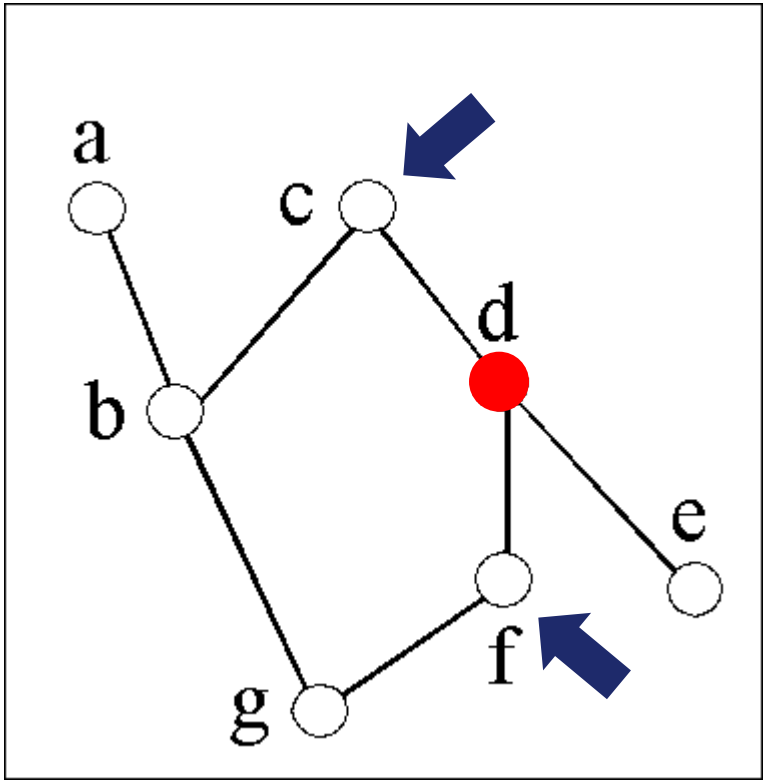
```
{leftmost outermost reduction}
  fst(square 4, undefined)
= {definition of fst}
  square 4
= {definition of square}
  4 x 4
= {definition of x}
 16
```

Lazy Evaluation

- Outermost terminiert häufiger
- Terminieren beide, so liefern sie das gleiche Ergebnis
- Wenn ein Ausdruck eine Normalform besitzt (Ausdruck der nicht weiter reduziert werden kann), kann Outermost sie berechnen
- Gleiche Ausdrücke müssen ggf. wiederholt berechnet werden
  - Lösung:Graphen

Graphenreduktion

- Zwei Knoten können den gleichen Kindknoten haben



Graphenreduktion

- Zeiger

{leftmost outermost reduction}

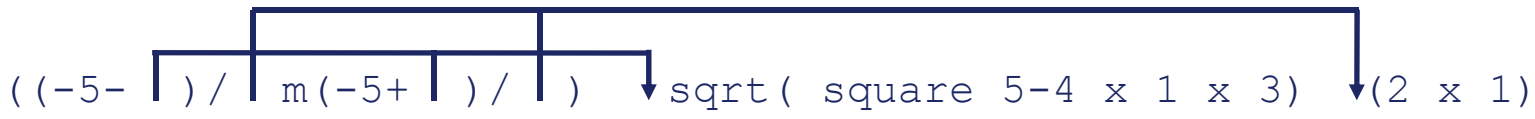
```
  square(3+4)
= {definition of square}

  (x)  (3+4)
= {definition of +}

  (x)  (7)
= {definition of x}
  49
```

Graphen

- Geteilte Ausdrücke sind ebenfalls in lokalen Definitionen zu finden

```
roots a b c = ((-b-d)/e,(-b+d)/e)
    where d = sqrt (square b-4 x a x c)
          e = 2 x a
```

- Der erste Reduktionsschritt für `roots 1 5 3`



Graphenreduktion

- Abschätzen der Größe des Graphen
- Knoten zählen heißt Parameter zählen
- Multiplikation (2 Parameter), Addition (2 Parameter)


```
square (3+4)
```

- Square (1 Parameter), Addition (2 Parameter)

Effizenz Lazy Evaluation

- geg.: eine Reduktionsfolge
- $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$
- Zeitbedarf für Reduktion: $\boldsymbol{n + \varepsilon}$
- ε entspricht dem Zeitaufwand der Reduktions-Suche
- Es wird die Zeit vernachlässigt, die zur Suche des Ausdrucks benötigt wird der den Ansprüchen einer äußersten Reduktion genügt. Bei großen Ausdrücken ein gewichtiger Faktor
- Platzbedarf für Reduktion: Größe des größten Ausdrucks
- Speicherplatz wird durch „Garbage Collection“ wiederholt genutzt

Effizenz Lazy Evaluation

- Lazy Evaluation wird in Haskell eingeführt, weil
  - Reduktionen terminieren, wenn es eine Reduktionsfolge gibt die terminiert
  - die Eager Evaluation genau so viele oder mehr Schritte benötigt um zu terminieren

Komplexität

Ω

Komplexität

- Zeitkomplexität (*time complexity)*
  - Wie viel Laufzeit benötigt ein Programm?
- Raumkomplexität (*space complexity)*
  - Wie viel Speicherplatz benötigt ein Programm?
- Die Komplexität eines Algorithmus oder einer Funktion bezeichnet die Wachstumsrate von Ressourcen wie zum Beispiel der Laufzeit gegenüber der Menge der zu verarbeitenden Daten

Komplexität

Notation	Definition
$f \in \mathrm{O}(g)$	Asymptotische obere Schranke
$f \in \Omega(g)$	Asymptotisch untere Schranke
$f \in \Theta(g)$	Asymptotisch scharfe Schranke

- „=“ nicht im herkömmlichen Sinne, eher „ist enthalten in“
- O, Ω, Θ definieren Mengen. Diese Mengen enthalten alle Funktionen für die eine Konstante C existiert und die die Ungleichung für O, Ω oder Θ erfüllen.

Komplexitäten

- Häufig anzutreffen:
- $O(1) = \Omega(1) = \Theta(1)$, konstantes Wachstum (Array-Zugriff)
- $O(\log n)$, logarithmisches Wachstum (binäres Suchen)
- $O(n)$, lineares Wachstum (Zugriff auf alle Elemente einer Liste)
- $O(n \log n)$, n-faches logarithmisches Wachstum (heap-sort)
- $O(n^k)$, polynomisches Wachstum (quadratisch, selection-sort)
- $O(2^n)$, exponentielles Wachstum

Warum Eager Evaluation?

- Unter Eager Evaluation kann die Laufzeitanalyse kompositionell betrachtet werden
- Es gilt weiterhin:
  - theoretische obere Grenzen für Eager & Lazy Evaluation identisch
  - häufig: untere Grenzen für Eager & Lazy Evaluation ebenfalls identisch

```
minlist = head . sort
```

```
T(minlist)(n) = T(sort)(n) + T(head)(n)
```

Laufzeitanalyse (Reverse)

```
reverse1 [] = []
reverse1 (x:xs) = reverse1 xs ++ [x]

reverse2 = foldl prefix []
  where prefix xs x = x : xs
```

- $O(n^2)$
- $O(n)$

```
{ Zur Erinnerung }
(++) :: [α] -> [α] -> [α]
  [] ++ ys = ys
  (x:xs) ++ ys = x : (xs ++ ys)

foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl op e [] = e
  foldl op e (x : xs) = foldl op (e `op` x) xs
```

Analyse reverse1

```
T(reverse1)(0)   = O(1)  =  Θ(1)
T(reverse1)(n+1) = T(reverse1)(n) + T(++)(n,1)
```

- Um eine Liste der Länge n+1 umzudrehen, dreht man eine Liste der Länge n um und konkateniert es mit einer Liste der Länge 1.
- Rekursion für T(reverse1) lösen und Einsatz der Gleichung: T(++)(n,m) = Θ(n)

$$T(reverse1)(n) = \sum_{i=0}^{n} \Theta(n) = \Theta(n^2)$$

Analyse reverse2

- Vorbereitende Maßnahme: foldl eliminieren, um direkte Rekursion zu erhalten

```
reverse2 = foldl prefix []
  where prefix xs x = x : xs

{ umformen, Hilfsfunktion definieren: }
reverse2 xs = accum [] xs
accum ws[] = ws
accum ws (x:xs) = accum (x:ws) xs
```

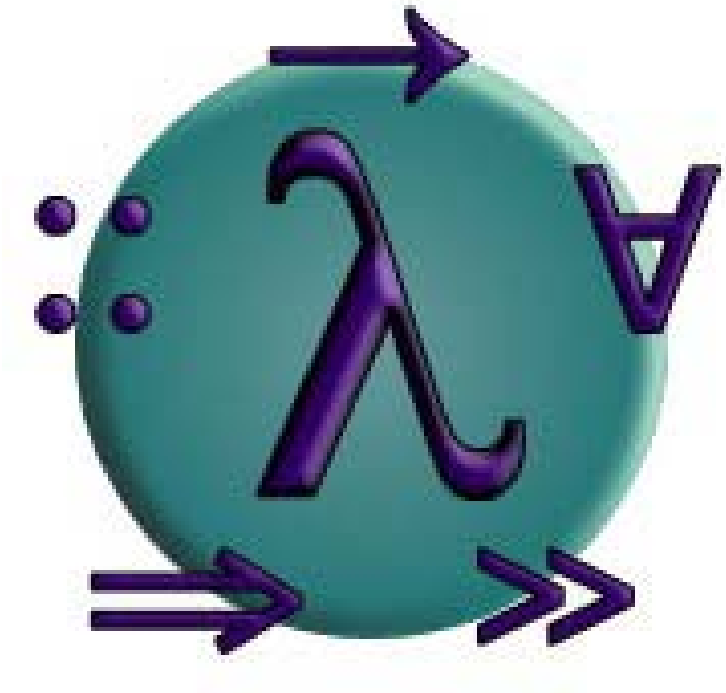
Analyse reverse2

- accum nimmt zwei Argumente auf
- Länge der Argumente: m, n

```
T(accum)(m,0) = O(1) = Θ(1)
T(accum)(m,n+1) = O(1) + T(accum)(m,n)

T(accum)(m,n) = Θ (n)
```

$$T(reverse2) = \Theta(n)$$

Parameterakkumulation



Zusätzlicher Parameter

- Durch Hinzufügen eines weiteren Parameters Laufzeit verbessern
- Häufig genutzte Technik um „teure“ ++-Operationen zu vermeiden

Beispiel flatten

```
{ Bekannt: }
flatten :: Btree a -> [a] -> [a]
flatten (Leaf x) = [x]
flatten (Fork xt yt)= flatten xt ++ flatten yt
```

```
{ Neue Definition }
flatcat :: Btree a -> [a] -> [a]
flatcat xt xs = flatten xt ++ xs
{ Es gilt: flatten xt = flatcat xt [] }

{ Rekursive Struktur für flatcat }
flatcat :: Btree a -> [a] -> [a]
flatcat (Leaf x) xs = x : xs
flatcat (Fork xt yt) xs = flatcat xt (flatcat yt xs)
```

Laufzeitanalyse zum Glätten eines perfekten binären Baums

- flatten

$$
\begin{aligned}
T(\text{flatten})(0) &= O(1) \\
T(\text{flatten})(h+1) &= 2T(\text{flatten})(h) + T(++)(2^h, 2^h) \\
T(\text{flatten})(h+1) &= 2T(\text{flatten})(h) + \Theta(2^h) \\
\\
T(\text{flatten})(h) &= \Theta(h2^h)
\end{aligned}
$$

- Aus Induktion folgt: $T(\text{flatten})(h) = \Theta(h2^h)$
- flatten benötigt also O(s log s) Schritte auf einem Baum mit Anzahl der Blätter *s*

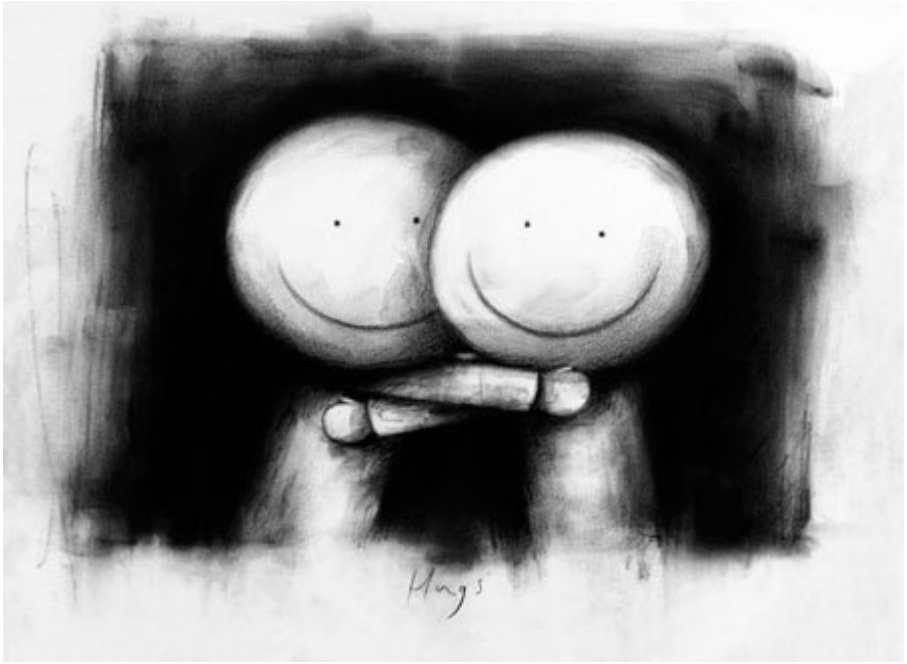
Laufzeitanalyse zum Glätten eines perfekten binären Baums

- Flatcat

```
T(flatcat)(0,n)    = O(1)
T(flatten)(h+1,n) = O(1) + T(flatcat)(h,2^h+n)+ T(flatcat)(h,n)
```

- T(flatcat)(h,n) h=Höhe des Baumes, n=Länge der Liste
- Aus Induktion folgt: $T(flatcat)(h,n) = \Theta(2^h)$
- flatcat benötigt also lineare Laufzeit O(s) im Verhältnis zur Anzahl der Blätter s

Tupling



Tupling

- Ein weiteres Ergebnis in Funktionen mitführen
- Umsetzung über Ergebnistupel
- Fibonacci

```
fib :: Integer -> Integer
```

```
fibtwo :: Integer -> (Integer,Integer)
```

Beispiel Fibonacci

```
{ Bekannt  }
fib 0 = 0
fib 1 = 1
fib (n+2) = fib n + fib(n+1)
```

```
{ Zeitkomplexität }
T(fib)(0) = O(1)
T(fib)(1) = O(1)
T(fib)(n+2) = T(fib)(n) + T(fib)(n+1) + O(1)
```

Tupling

```
fibtwo n = (fib n, fib(n+1))

{Direkte Rekursion:  }
fibtwo 0 = (0,1)
fibtwo (n+1) = (b,a+b)
  where (a,b) = fibtwo n

T(fib) = Θ(n) { lineare Laufzeit }
```

- Laufzeit von exponentiell zu linear!

Tupling

```
{ berechnet den Durchschnitt einer Float-Liste }
average :: [Float] ->Float
average xs = (sum xs) / (length xs)
```

- Zwei Traversierungen nötig

```
sumlen xs = (sum xs, length xs)
```

```
{ Direkte Rekursion }
sumlen [x] = (x,1)
sumlen (x:y:xs) = (x+z,n+1)
  where (z,n) = sumlen(y:xs)
```

```
average' = uncurry(/) . Sumlen
```

- Eine Traversierungen nötig

Anmerkungen zu average‘

- Beide Funktionen haben eine Laufzeit Θ(n)
- Der Zeitgewinn durch die einmalige Traversierung, könnte durch die Bildung von Ergebnistupeln wieder verloren gehen
- Warum also average‘?
- Speicherplatz einsparen mit average‘
- `average [1..1000]` – da average xs zweimal auf der rechten Seite xs referenziert, wird der doppelte Speicherplatz benötigt
- Mit der zweiten Definition von average wird eine konstante Belegung des Speichers erreicht

Speicherplatz

Reduktion & Speicherplatz

- Reduktionsfolge sum [1..1000]

```
sum = foldl (+) 0
```

```
sum [1..1000]
 = foldl (+) 0 [1..1000]
 = foldl (+) (0+1) [2..1000]
 = foldl (+) ((0+1)+2) [3..1000]
 :
 :
 = foldl (+) (…((0+1)+2)+…+1000)[]
 = (…((0+1)+2)+…+1000)= 500500
```

- Die Berechnung von sum [1..n] mit *outermost reduction* wächst proportional zur Größe von n

Neue Reduktionsstrategie

```
sum [1..1000]
 = foldl (+) 0 [1..1000]
 = foldl (+) (0+1) [2..1000]
 = foldl (+) 1 [2..1000]
 = foldl (+) (1+2) [3..1000]
 = foldl (+) 3 [3..1000]
 :
 :
 = foldl (+) 500500 []
 = 500500
```

- Mischung aus *innermost* und *outermost reduction*
- Neue Funktion Definieren, die Kontrolle über Reduktion erlaubt

Strict Funktion

```
strict f x = if x =  ⊥ then ⊥  else f x
```

- Kontrolliert die Reduktionsfolge
- in Term `strict f e`, reduziert `e` auf Kopf Normal-Form
- Jeder Ausdruck in Normalform ist auch in Kopf Normalform, aber nicht umgekehrt (Normalform kann nicht mehr reduziert werden)

Strict Funktion

```
inc x = x+1
```

```
inc(inc(8x5))
 = inc(8x5)+1
 = ((8x5)+1)+1
 = (40+1)+1
 = 41+1
 = 42
```

```
strict inc(strict inc(8x5))
 = strict inc(strict inc(40))
 = strict inc(inc(40))
 = strict inc(40+1)
 = strict inc 41
 = inc 41
 = 41+1
 = 42
```

Strict Funktion

```
{ keine HUGS kompatible Notation }
sfoldl (⊕) a [] = a
sfoldl (⊕) a (x:xs)= strict (sfoldl (⊕)) (a⊕x) xs
```

```
sum = sfoldl(+) 0

sum[1..1000]
 = sfoldl (+) 0 [1..1000]
 = strict (sfoldl(+))(0+1) [2..1000]
 = sfoldl (+) 1 [2..1000]
 = strict (sfoldl(+))(1+2) [3..1000]
 = sfoldl (+) 3 [3..1000]
 :
 :
 = sfoldl (+) 500500 []
 = 500500
```

Fazit

Fazit

- Anwenden von einfachen Konzepten kann große Wirkung erzielen
- Durch das Verständnis innerer Strukturen können Algorithmen effizienter gestaltet werden

Vielen Dank für Eure Aufmerksamkeit

Literatur

- Richard Bird
  Introduction to Functional Programming using Haskell
  Second Edition 1998
  Prentice Hall
  ISBN 0-13-484346-0
- http://www.techfak.uni-bielefeld.de/ags/pi/lehre/AuDIWS05/