Effizienz in Haskell

Informatikseminar WS03/04

Oliver Lohmann – mi4430

Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

Definition Effizienz

- "Verhältnis zwischen dem erzielten Ergebnis und den eingesetzten Mitteln " (*ISO 9000:2000*)

💣Nicht zu verwechseln mit der Defintion von *Effektivität*:

- "Ausmaß, in dem geplante Tätigkeiten verwirklicht und geplante Ergebnisse erreicht werden" (*ISO 9000:0000*)

Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

Beispiel zu Auswertungsschemata I

Innerste Reduktion
innermost reduction

```
   square (3+4)
=  {definition of +}
   square (7)
=  {definition of square}
   7 × 7
=  {definition of ×}
   49
```

Äußerste Reduktion
outermost reduction

```
   square (3+4)
=  {definition of square}
   (3+4) × (3+4)
=  {definition of +}
   7 × (3+4)
=  {definition of +}
   7 × 7
=  {definition of ×}
   49
```


Beispiel zu Auswertungsschemata II

Innerste Reduktion
innermost reduction

```
   fst (square 4,square 2)
=  {definition of square}
   fst(4 × 4, square 2)
=  {definition of × }
   fst(16, square 2)
=  {definition of square}
   fst(16, 2 × 2)
=  {definition of ×}
   fst(16,4)
=  {definition of fst}
   16
```

Äußerste Reduktion
outermost reduction

```
   fst(square 4,square 2)
=  {definition of fst}
   square 4
=  {definition of square}
   4 × 4
=  {definition of ×}
   16
```

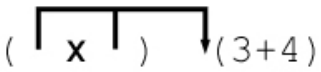
Äußerste-/Innerste Reduktion

- Äußerste Reduktion terminiert häufiger
- Wenn sie terminieren, liefern beide Strategien das gleiche Ergebnis
- Vorteil Äußerste Reduktion:
  - Falls eine Normalform vorhanden ist, kann sie berechnet werden
  - ↳ Äußerste Reduktion wird auch *Normale Ordnung* (*normal order*) bezeichnet
- Nachteil Äußerste Reduktion:
  - Gleiche Ausdrücke in Reduktionen müssen wiederholt berechnet werden (Bsp. I)
  - ↳ Lösung: Graphen

Graphen

- Bäume wurden definiert als eine Menge von durch Kanten miteinander verbundenen Knoten, wobei die Kanten gerichtet von der Wurzel zu den Blättern verlaufen. Zusätzlich durfte jeder Knoten nur maximal einen Vorgängerknoten besitzen (spezielle Form der Graphen)
- Bei Graphen entfallen diese Einschränkungen: ein Graph besteht aus Knoten, die durch Kanten verbunden sind. Die Kanten können dabei entweder gerichtet oder ungerichtet sein.

Graphen



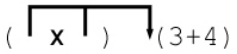
Entspricht:

```
(3+4) × (3+4)
```

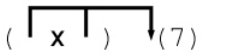
↳ Äußerste Graphen Reduktion

outermost graph reduction

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




```
=  {definition of +}
```




```
=  {definition of ×}
   49
```

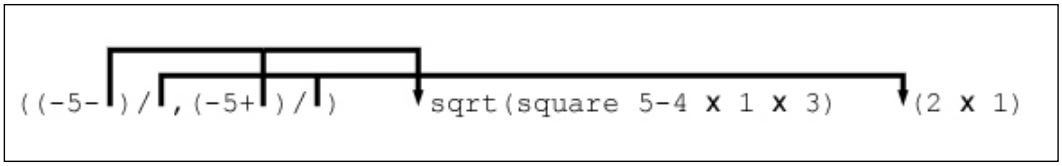
Nur drei Reduktionsschritte!

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 × a × c)
                      e = 2 × a
```

↳ Der erste Reduktionsschritt für `roots 1 5 3`



Effizienz Lazy Evaluation

geg.: eine Reduktionsfolge

$e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \ldots \Rightarrow e_n$

↳ Zeitbedarf für Reduktion:

- $n + \varepsilon$, 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)

Zusammenfassung

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

Ausdrücke werden in Haskell grundsätzlich nicht strikt ausgewertet!

Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

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 Resourcen wie zum Beispiel der Laufzeit gegenüber der Menge der zu verabeitenden Daten.

Ordnung (*order notation*)

- Mathematisches Verfahren zur Einordnung von Funktionen:
  - geg.: zwei Funktionen *f*, *g*
    eine positive Konstante *C*
    eine natürliche Zahl n_0

Hinweis: Funktionsresultate nur für große n aussagekräftig.

1. O-Notation (*big O-notation*)

f hat **höchstens** den Grad *g*:
`f = O(g)`, wenn gilt: `f(n) ≤ C g(n)` für alle `n ≥ n₀`

Ordnung (*order notation*)

2. Ω-Notation (big Omega-*notation*)

f hat **mindestens** den Grad *g*:
`f = Ω(g)`, wenn gilt: `f(n) ≥ C g(n)` für alle `n ≥ n₀`

Führt man beide Regeln zusammen, erhält man ein drittes Kriterium:

3. Θ-Notation (big Theta-*notation)*

f hat **exakt** den Grad *g*:

Anmerkungen *order-notation*

- „=“ 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.

- Häufig anzutreffen:
  - O(1) = Ω (1) = Θ(1), konstantes Wachstum (Array-Zugriff)
  - $O(\lg_n)$, 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

Hinweise

$O(n)$ beschreibt das Verhalten einer anonymen Funktion f

$O(n)+O(n)=O(n)$, da $f_1(n)+f_2(n)=O(n)$

Ein Ausdruck wie $\sum_{n=1}^{N} O(n^2)$ bedeutet nicht :

$O(1^2)+O(2^2)+\ldots+O(N^2)$

Statt dessen referenziert $O(n^2)$ eine einzelne Funktion f,
die den Anprüchen $f(n)=O(n^2)$ genügt.

$$\sum_{n=1}^{N} f(n) \Rightarrow \sum_{n=1}^{N} O(n^2) = O(N^3)$$

Beispielrechnung

- geg.:
  - eine Liste
  - eine Funktion *append*, fügt Elemente hinten an die Liste an

  Berechnung:
  - Ein Element anhängen:
    O(n), Liste mit n-Elementen traversieren

  - n Elemente anhängen:
    1 + 2 + 3 + … n-1 + n, entspricht:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

Zeitkomplexität

Notation: T(f)(n)

Drei wichtige Hinweise:

1. Auf die Notation der Speicherplatzkomplexität wird erst im weiteren Verlauf genauer eingegangen, hier noch nicht formal korrekt.
2. T(f)(n) ist stets eine *worst-case* Einschätzung.
3. Die *Eager Evaluation* wird als Auswertungsgrundlage genutzt.

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

Beispiel *reverse*

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

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

Zur Erinnerung:

```
(++)           :: [α] → [α] → [α]
[] ++ ys       = ys
(x:xs) ++ ys   = x : (xs ++ ys)
```

Analyse *reverse1*

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

Aussage 2. Gleichung:
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(reverse2) = Θ(n)
```


Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

Idee

- 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*

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

Aus Induktion folgt:
$T(flatten)(h) = \Theta(h2^h)$
`flatten` benötigt also $O(s \log s)$ Schritte auf einem Baum der Größe *s*.

- *flatcat*

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

Aus Induktion folgt:
$T(flatcat)(h,n) = \Theta(2^h)$
`flatcat` benötigt also lineare Laufzeit im Verhältnis zur Baumgröße.

Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

Idee

- Ein weiteres Ergebnis in Funktionen mitführen
  ↳ Umsetzung über Ergebnistupel, z.B.:

statt:

```
fib          ::    Integer → Integer
```

besser:

```
fibtwo       ::    Integer → (Integer,Integer)
```


Beispiel *fib*

Bekannt:

```
fib 0          = 0
fib 1          = 1
fib (n+2)      = fib n + fib(n+1)
```

- Laufzeitanalyse für *fib*

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

Aus Induktion folgt:
`T(fib)(n)` = Θ`(fib n)`, benötigte Zeit ist also proportional zur Größe des Ergebnisses.
`fib n` = $\Theta(\Phi^n)$, mit Φ = `(1+sqrt(5))/2` (Goldener Schnitt)

exakt: $fib(n) = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n = \frac{1}{\sqrt{5}} \cdot \Phi^n$

Beispiel *fib*

```
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

Steigerung der Effizienz von exponentiell zu linear!

Beispiel *average*

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

↳ Zwei Traversierungen nötig

```
-- berechnet Summe und Länge einer (Float-)Liste
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)
```

```
-- berechnet den Durchschnitt einer Float-Liste
average'            = uncurry(/) . sumlen
```

↳ Eine Traversierung nötig

Anmerkungen zu *average'*

- Beide Funktionen haben eine Laufzeit $\Theta(n)$
- Der Zeitgewinn durch die einmalige Traversierung, könnte durch die Bildung von Ergebnistupeln wieder verloren gehen

↳ Warum also *average'* ???

💾 Speicherplatz sparen!

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 erreichen wir eine konstante Belegung des Speichers

Gliederung

- Allgemeine Definition von Effizienz
- Lazy Evaluation
- Asymptotische Analyse
- Parameter Akkumulation
- Tupling
- Speicherplatz kontrollieren

Reduktionen & Speicherplatz

- Reduktionsfolge `sum [1..1000]` mit `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* & *outermost reduction*

↳ Neue Funktion Definieren, die Kontrolle über Reduktion erlaubt

strict Funktion

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

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

Idee: *foldl* als *strict* definieren

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

Mit `sum = sfoldl(+) 0`, entsteht eine neue Reduktionsfolge:

```
      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
```


Verbindung mit *average* herstellen

```
average'    = uncurry (/) . Sumlen
sumlen      = sfoldl f (0,0)
            where f (s,n) x = (s+x, n+1)
```

- Es halten wir folgende Programmreduktionsfolge unter Speicherplatznutzung?

```
      sumlen [1..1000]
=     sfoldl f (0,0) [1..1000]
=     strict (sfoldl f) (0+1,0+1) [2..1000]
=     sfoldl f (0+1,0+1) [2..1000]
=     strict (sfoldl f) ((0+1)+2,(0+1)+1) [3..1000]
=     …
```

↯ Nein, da die Hilfsfunktion f nicht strikt ist.

Lösung *average* Problem

- *f* muss ebenfalls strikt definiert sein

```
f'(s,n) x = strict (strict tuple (s+x))(n+1)
            where tuple a b = (a,b)
```

Mit `f'`, entsteht die Reduktionsfolge:

```
  sum[1..1000]
= sfoldl f' (0,0) [1..1000]
= strict (sfoldl f') (strict(strict tuple(0+1))(0+1)) [2..1000]
= strict (sfoldl f') (strict(strict tuple(0+1))1) [2..1000]
= strict (sfoldl f') (strict(strict tuple 1)1) [2..1000]
= strict (sfoldl f') (1,1) [2..1000]
= sfoldl f' (1,1) [2..1000]
```


Anmerkungen zu *fold*

- wenn (⊕) assoziativ zu der Identität *e* ist, dann gilt:

  `foldr (⊕) e xs = foldl (⊕) e xs,`
  für alle endlichen Listen xs

↳ Es bestehen Unterschiede in der Komplexität!

„Daumenregel“:
Wenn (⊕) nicht strikt ist, dann ist foldr häufig effizienter!

Beispiele: and und ++.

Ende.