Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 21.01.09: Effizienzaspekte

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke
  - Datenmodellierung mit XML
  - Effizienzerwägungen
  - Grafik
  - Schluss

Inhalt

- Zeitbedarf: Endrekursion — `while` in Haskell
- Platzbedarf: Speicherlecks
- "Unendliche" Datenstrukturen
- Verschiedene andere Performancefallen:
  - Überladene Funktionen, Listen

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

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

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: `if` oder `case`
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in Sprung oder Schleife übersetzt.
- Nur nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` nicht endrekursiv:

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

- `fac` endrekursiv:

```
fac :: Integer-> Integer
fac n      = fac0 n 1 where
  fac0 :: Integer-> Integer-> Integer
  fac0 n acc = if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` 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..20000])`

Überführung in Endrekursion

- Gegeben Funktion

```
f' : S -> T
f' x = if B x  then H x
               else φ (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 -> T
f x = g x e where
   g x y = if B x  then φ (H x) y
                   else g (K x) (φ (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:

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

- 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)
```

“Unendliche” Listen

- Listen müssen nicht endlich repräsentierbar sein:
  - Beispiel: definiert “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]
    cycle xs = xs ++ cycle xs
    ```

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

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 (vgl. alte Version)

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

- Von allen Primzahlen die ersten:

```
nprimes :: Int-> [Integer]
nprimes n = take n primes
```

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: exponentieller Aufwand.

Bsp: Fibonacci-Zahlen

- Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
            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.

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`, `nat` nicht mit darstellbar
- Damit beispielsweise auch Graphen modellierbar

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

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!

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Erzwungene Striktheit:
  - `seq :: a-> b-> b` erzwingt Striktheit im ersten Argument:

    $\bot$ `‘seq‘ b = ⊥`

    `a ‘seq‘ b = b`
  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung
  - `ghc` macht Striktheitsanalyse

```
f $! x = x ‘seq‘ f x
```

- Fakultät in konstantem Platzaufwand

```
fac2 n = fac0 n 1 where
  fac0 n acc = seq acc $ if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

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' :: (a-> b-> a)-> a-> [b]-> a` ist strikt, endrekursiv.
- Für Monoid (ϕ, e) gilt

$$\texttt{foldr}\ \phi\ e\ l = \texttt{foldl}\ (\texttt{flip}\ \phi)\ e\ l$$

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
```

  - Unendliche Listen immer mit `foldr` falten.

Gemeinsame Teilausdrücke

- Ausdrücke werden intern durch Termgraphen dargestellt.
- Argument wird nie mehr als einmal ausgewertet:

```
f :: Int-> Int-> Int
f x y = x+ x
```

  - Beispiel: `f (trace "Eins" (3+2)) (trace "Zwei" (2+7))}`

- Sharing von Teilausdrücken
  - Explizit mit `where` und `let`
  - Implizit (`ghc`, optimierend):

```
double :: Int-> Int
double x = trace "Foo!" 2*x
```

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.
  - 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`

Zusammenfassung

- Endrekursion: `while` für Haskell.
  - Überführung in Endrekursion meist möglich.
  - Noch besser sind strikte Funktionen.
- Speicherlecks vermeiden: Striktheit und Endrekursion
- Datenstrukturen müssen nicht endliche repräsentierbar sein
- Überladene Funktionen sind langsam.
- Listen sind keine Felder oder endliche Abbildungen.
- Effizienz muss nicht immer im Vordergrund stehen.