Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 30.01.2007: Effizienzaspekte

Christoph Lüth

WS 06/07

Inhalt

- *Zeitbedarf*: *Endrekursion* — `while` in Haskell
- *Platzbedarf*: *Speicherlecks*
- 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 Haskell
  - *Striktheit*: *Speicherlecks* vermeiden
- 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. (Zeigen)

Beispiele

- 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 $O(n)$: (Zeigen)

```
rev :: [a]-> [a]
rev xs = rev0 xs [] where
    rev0 []     ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```


Überführung in Endrekursion

- Gegeben eine Funktion f' f': S-> T

```
f' x = if B x  then H x
               else φ (f' (K x)) (E x)
```

  - Mit K:S-> S, ϕ:T-> T-> T, E:S-> T, H:S-> T.
- Sei ϕ assoziativ, e:T neutrales Element
- Dann ist die 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:

$$\begin{array}{rcl@{\qquad}rcl} K(x) & \mapsto & \texttt{tail} & B(x) & \mapsto & \texttt{null}\ x \\ E(x) & \mapsto & 1 & H(x) & \mapsto & 0 \\ \phi(x,y) & \mapsto & x+y & e & \mapsto & 0 \end{array}$$

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

- Damit ergibt sich 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]

instance Show a=> Show (Set a) where
  showsPrec i (Set elems) =
     \r-> "{" ++ concat (intersperse ", "
                         (map show elems)) ++ "}" ++ r
```

Nicht *perfekt* — 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*
- Eine Funktion hat ein *Speicherleck*, wenn Speicher *unnötig* lange im Zugriff bleibt.
  - "Echte" Speicherlecks wie in C/C++ *nicht möglich*.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden *nicht auswertet*. (Zeigen.)
  - Insbesondere ärgerlich bei *nicht-terminierenden Funktionen*.

Striktheit

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

```
⊥ `seq` b = ⊥
a `seq` b = b
```

  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung

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

- Fakultät in konstantem Platzaufwand (zeigen):

```
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'` *endrekursiv* und *konstanter Platzaufwand*, aber traversiert immer die *ganze* Liste.
- Wann welches `fold`?
  - *Strikte Funktionen* mit `foldl'` falten.
  - 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: (Zeigen)

```
f :: Int-> Int-> Int
f x y = x+ x
test1 = f (trace "Eins" (3+2)) (trace "Zwei" (2+7))
list1 =   [trace "Foo" (3+2), trace "Bar" (2+7)]
```

- *Sharing* von Teilausdrücken
  - Explizit mit `where` und `let`
  - Implizit (`ghc`)

Memoisation — *trading space for time*

- *Fibonacci-Zahlen* als Strom: *linearer Aufwand*.

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

- *Kleine Änderung*: "unnötiger" Parameter ()

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                (tail (fibsFn ()))
```

- *Große Wirkung:* Exponentiell. *Warum?*
- Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.

Die Abhilfe: Memoisation

- *Memoisation*: Bereits berechnete Ergebnisse speichern.
- In Hugs: Aus dem Modul `Memo`:

```
memo :: (a-> b)-> a-> b
```

- Damit ist alles wieder gut:

```
fibsFn' :: () -> [Integer]
fibsFn' = memo (\()-> 1 : 1 : zipWith (+)
                              (fibsFn' ())
                              (tail (fibsFn' ())))
```

- `GHC` *kann* das *automatisch*.

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber *langsam*.
  - Implementierung von Typklassen: *Verzeichnis* (dictionary) von Klassenfunktionen.
  - Überladung wird zur *Laufzeit* aufgelöst.
- Bei kritischen Funktionen durch Angabe der Signatur *Spezialisierung erzwingen*.
- 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.
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit.
  - Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- Felder:
  - Feste Länge
  - Zugriff auf n-tes Element in konstanter Zeit.
  - Abstrakt: Abbildung Index auf Daten

Felder in Haskell

- Modul `Array` aus der Standardbücherei

```
data Ix a=> Array a b  -- abstract
array     :: (Ix a) => (a,a) -> [(a,b)]
                                  -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
(//)      :: (Ix a) => Array a b -> [(a,b)]
                                  -> Array a b
```

- Als Indexbereich geeignete Typen (Klasse `Ix`): `Int`, `Integer`, `Char`, `Bool`, Tupel davon, Aufzählungstypen.

Listen von Paaren sind keine endlichen Abbildungen.

- `Data.Map`: *endliche Abbildungen*
  - Effizient, weil durch balancierte Bäume implementiert.
- Leere Abbildung:

```
empty :: Ord k => Map k a
```

- Hinzufügen:

```
insert     :: Ord k => k -> a -> Map k a -> Map k a
insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a ->
```

  - Verhalten bei Überschreiben kann spezifiziert werden.
- Auslesen:

```
lookup :: Ord k => k -> Map k a -> Maybe a
findWithDefault :: Ord k => a -> k -> Map k a -> a
```


Zusammenfassung

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