Kapitel 8: Effizienzaspekte

Lernziele dieses Kapitels

1. Rolle der Effizienz bei der funktionalen Programmierung
2. Funktionale Arrays: mutable/immutable, Laziness, mit IO/ST-Monade
3. Effiziente funktionale Datenstrukturen können besser als Arrays sein
4. Asymptotische Komplexität ist wichtig (Effizienzanalyse)
5. Bewusster Umgang mit Speicherplatz,
   Verwendung von Striktheitsannotationen

Rolle der Effizienz

- Funktionales Programm im Hochleistungsrechnen
  - nicht geeignet, um daraus mit Standardcompiler (ghc) Zielcode zu erzeugen (hier kommen nur Sprachen wie C oder Fortran in Frage)
  - geeignet als Metaprogramm, um paralleles Programm zu erzeugen
    - Beherrschung komplexer Kommunikationsmuster
    - Vermeidung von Redundanz
    - Konzentration von maschinennahen Aspekten
- Motivation für Effizienzbetrachtungen
  - Ausreichende Geschwindigkeit für nicht zeitkritische Anwendungen
  - Speicherverbrauch unterhalb der Stack- bzw. Heapgröße halten

Verwendung von Arrays

- geeignet für sehr große Datenmengen
- Anforderungen
  - Indexmenge effizient auf `Int` abbildbar
  - homogener Elementtyp
  - kein Bedarf an Sharing zwischen Versionen (vor/nach Feldänderung)
- Anwendungsdomänen
  - Bildverarbeitung
  - Elektrotechnik: Vektoren und Matrizen
  - relationale Datenbanken
  - Operations Research: Matrizen für Kosten von Zuordnungen

Sinnvoller Einsatz von Arrays

Arrays sollten nur dort eingesetzt werden, wo sie geeignet sind

Viele Algorithmenbücher verwenden Arrays als Datenstruktur aus anderen Gründen:

- Schleifenprogramme mit Arrays sind für Anfänger in fast jeder imperativen Programmiersprache ohne Anstrengungen implementierbar
- Algorithmus stammt aus einer Zeit, zu der dynamische Speicherverwaltung nicht weit verbreitet war

In einer funktionalen Sprache kann die Verwendung einer verketteten Struktur einfacher und effizienter sein als die eines Arrays!
Bsp.: Heaps gespeichert in einem Baum statt einem Array

Typklasse Ix (1)

verwendet zur Indizierung, insbesondere von Arrays

```
class  (Ord a) => Ix a  where
   range     :: (a,a) -> [a]
   index     :: (a,a) -> a -> Int
   inRange   :: (a,a) -> a -> Bool
   rangeSize :: (a,a) -> Int
```

Eigenschaften, die Implementierung haben soll:

```
range   (l,u)  !!  index (l,u) i  ==  i          -- when i is in range
inRange (l,u) i                   ==  i `elem` range (l,u)
```

Typklasse `Ix` (2)

Beispiel:

```
data Colour = Red | Orange | Yellow | Green | Blue | Indigo | Violet

range   (Yellow,Blue)         ==  [Yellow,Green,Blue]
index   (Yellow,Blue) Green   ==  1
inRange (Yellow,Blue) Red     ==  False
```

Vordefinierte Instanzen von `Ix`:

- `Char`, `Int`, `Integer`, `Bool`, ...
- `(Ix a, Ix b) => Ix (a, b)`

...

Typklasse Array

- Arraytyp mit Indextyp a und Elementtyp b:

```
data  (Ix a)    => Array a b = ... -- Abstract
```

- Erzeugung eines Arrays

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

```
> array (0,1) [(1,2),(0,5)]
array (0,1) [(0,5),(1,2)]
> listArray (0,3) ['a'..'z']
array (0,3) [(0,'a'),(1,'b'),(2,'c'),(3,'d')]
```

- Indizierung

```
(!) :: (Ix a) => Array a b -> a -> b
Array> listArray ('a','z') [0..] ! 'c'
2
```

- Arraygrenzen

```
bounds :: (Ix a) => Array a b -> (a,a)
> bounds (listArray ('a','z') [0..])
('a','z')
```

- Indizes, Elemente, Assoziationsliste

```
indices :: (Ix a) => Array a b -> [a]
elems   :: (Ix a) => Array a b -> [b]
assocs  :: (Ix a) => Array a b -> [(a,b)]
```

- Arrayelemente ändern (update)

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

```
> listArray (0,3) ['a'..'z'] // [(2,'#'),(3,'%')]
array (0,3) [(0,'a'),(1,'b'),(2,'#'),(3,'%')]
```

Array-Kombinatoren

- Transformation der Elementmenge

```
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b
instance Functor (Array a) where ...
> fmap toUpper (listArray (0,3) ['a'..'z'])
array (0,3) [(0,'A'),(1,'B'),(2,'C'),(3,'D')]
```

- Transformation der Indexmenge

```
ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b)
                         -> Array b c -> Array a c
> ixmap ('a','d') (\i-> ord i - ord 'a')
                  (listArray (0,3) [2,3,5,7])
array ('a','d') [('a',2),('b',3),('c',5),('d',7)]
```

Akkumulierende Array-Generierung

- initial

```
accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a)
                        -> [(a,c)] -> Array a b

> let arr = accumArray (+) 0 (False,True)
            [(False,2),(True,1),(False,1)]
> arr
array (False,True) [(False,3),(True,1)]
```

- basierend auf bestehendem Array `arr`

```
accum :: (Ix a) => (b -> c -> b) -> Array a b
                       -> [(a,c)] -> Array a b

> accum (+) arr [(False,2),(True,5)]
array (False,True) [(False,5),(True,6)]
```

Zweidim. Arrays, Bsp.: Matrix-Multiplikation

```
newtype Matrix a b = M (Array (a,a) b) deriving Show

matmult :: Num b => Matrix Int b -> Matrix Int b
                                 -> Matrix Int b
matmult (M x) (M y) =
 let ((z0,z1),(m,n))  = bounds x
     ((z2,z3),(n',p)) = bounds y
 in if n/=n' || any (/=0) [z0,z1,z2,z3]
       then error "falsche Indexmenge"
       else M $ array ((0,0),(m,p))
                      [ ((i,j), sum [ x!(i,k) * y!(k,j)
                                    | k<-[0..n] ])
                      | i<-[0..m], j<-[0..p] ]
```

```
> a
M (array ((0,0),(2,1)) [((0,0),0),((0,1),1),
                        ((1,0),2),((1,1),3),
                        ((2,0),4),((2,1),5)])

> b
M (array ((0,0),(1,3))
  [((0,0),0),((0,1),1),((0,2),2),((0,3),3),
   ((1,0),4),((1,1),5),((1,2),6),((1,3),7)])

> matmult a b
M (array ((0,0),(2,3))
  [((0,0), 4),((0,1), 5),((0,2), 6),((0,3), 7),
   ((1,0),12),((1,1),17),((1,2),22),((1,3),27),
   ((2,0),20),((2,1),29),((2,2),38),((2,3),47)])
```

Laziness in Arrays

- Anwendung Tabulierung: einmal berechnete Werte werden wiederverwendet.
- Jedes Feld wird dann berechnet, wenn es zum ersten Mal gebraucht wird.

```
binom :: (Int,Int) -> Integer
binom (n,k) = let f n k | k==0        = 1
                        | n==k        = 1
                        | 0<k && k<n =   bs!(n-1,k-1)
                                       + bs!(n-1,k)
                        | otherwise   = 0
                  bs = array ((0,0),(n,k))
                       [((i,j),f i j) | i<-[0..n],
                                        j<-[0..k]]
              in bs!(n,k)
```

Eigenschaften von Haskell-Arrays

- Lesezugriff in konstanter Zeit
- Schreibzugriff in konstanter Zeit bei Verzicht auf Persistenz (Erhalt des alten Arrays)
- minimaler Speicherplatzverbrauch bei ungeboxten Elementtypen
- viele Möglichkeiten zur Indizierung (Typklasse `Ix`)
- intentionale Elementdefinition (Komprehensionen)
- akkumulierende Elementdefinition (Anwendung: Statistik)
- rekursive Definition von Elementen durch Benutzung bereits definierter Elemente

ghc-Arrays: Verwendungsart

- als normaler funktionaler Wert [default], Modul `Data.Array`
- Zugriff durch monadische Operationen
  - in der IO-Monade (bzw. IOS-Monade), Modul `Data.Array.IO`
  - in der im ghc vordefinierten ST-Monade, Modul `Data.Array.ST`

ghc-Arrays: updates

Array-Arten, die sich bzgl. des Updates unterscheiden

- immutable [default] (Instanz von `IArray`)
  - Persistenz des Arrays vor einem Update
  - neues Array nach dem Update is eine modifizierte Kopie
- mutable (Instanz von `MArray`)
  - Update eines Elements in konstanter Zeit, aber zerstört altes Array
  - bedingt monadische Operationen
  - notwendig beim Datenaustausch mit Programmen anderer Sprachen (C)
- diff-Arrays
  - mutable mit immutable-Interface
  - schneller Zugriff auf aktuelle Version des Arrays
  - alte Versionen verfügbar, aber mit langsamerem Zugriff

Repräsentation von Array-Elementen

- boxed [default]
  - Zeiger auf Elemente gespeichert
  - sinnvoll bei komplexen Strukturen mit Sharing
- unboxed
  - Elemente direkt hintereinander gespeichert
  - strikt in den Elementen
  - effizienter bezüglich Zeit- und Speicherverbrauch

Die im ghc vordefinierte ST-Monade

```
module ST( module ST, module Monad ) where
import Array
import Monad
import Control.Monad.ST
import Data.Array.ST
{-
 in Control.Monad.ST:
    data ST s a
    runST :: (forall s . ST s a) -> a
 in Data.Array.ST:
    Data STArray s i e
       importiert aus Data.Array.MArray (überladenes Interface):
    newArray   :: (MArray a e m, Ix i) => (i, i) -> e -> m (a i e)
    readArray  :: (MArray a e m, Ix i) => a i e -> i -> m e
    writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()  -}
```

Hoares quicksort-Algorithmus

```
quicksort :: Ord a => (Int,Int) -> STArray st Int a
                                -> ST st ()
quicksort (lower,upper) arr
 = if lower>=upper
      then return ()
      else do
           pivot <- readArray arr lower
           (i,j) <- partition arr pivot (lower,upper)
           quicksort (lower,j) arr
           quicksort (i,upper) arr
```

Verwendung von runST

```
sorttest :: Ord a => [a] -> [a]
sorttest xs =
 runST (do
         let n = length xs
         arr <- newArray (0,n-1) undefined
         mapM (\i -> writeArray arr i (xs!!i)) [0..n-1]
         quicksort (0,n-1) arr
         ys <- mapM (readArray arr) [0..n-1]
         return ys)
```

```
partition :: Ord a => STArray st Int a -> a -> (Int,Int)
                                        -> ST st (Int,Int)
partition arr pivot
 = while (\ (i,j) -> return (i<=j))
         (\ (i,j) -> do {
    i <- while (\i -> do x <- readArray arr i
                         return (x < pivot))
               (\i -> return (i+1)) i ;
    j <- while (\j -> do x <- readArray arr j
                         return (x > pivot))
               (\j -> return (j-1)) j ;
    if (i<=j) then (do xi <- readArray arr i
                       xj <- readArray arr j
                       writeArray arr i xj
                       writeArray arr j xi
                       return (i+1,j-1))
              else return (i,j) } )
```

while in der ST-Monade

```
while ::    (a -> ST st Bool)  -- Eintrittsbedingung
         -> (a -> ST st a)     -- Schleifenrumpf
         -> a
         -> ST st a
while p f x = do
                cont <- p x
                if cont then do
                               x <- f x
                               while p f x
                        else return x
```

Effizienzvergleich Quicksort

	15000	30000	60000
mit ST-Monade	2.4	14.5	63.5
`quickSortFast`	0.4	1.5	5.0

```
quickSortFast :: Ord a => [a] -> [a]
quickSortFast xs = qs xs [] where
   qs []     acc = acc
   qs (x:xs) acc = let (ys,zs) = partition (<x) xs
                   in qs ys (x : qs zs acc)
```

Effiziente funktionale Datenstrukturen

Beispiel: HeapTrees

Anwendungen: Prioritätswarteschlange, Heapsort

Eigenschaft: der Wert jedes Knotens ist nicht größer als jeder seiner Nachkommen

```
module SigHeap where
 class Heap t where
   empty     :: Ord a => t a
   isEmpty   :: Ord a => t a -> Bool
   minElem   :: Ord a => t a -> a
   deleteMin :: Ord a => t a -> t a
   insert    :: Ord a => a -> t a -> t a
   merge     :: Ord a => t a -> t a -> t a
   fromHeap  :: Ord a => t a -> [a]
   fromHeap h = if isEmpty h then []
                             else minElem h : fromHeap (deleteMin h)
```

```
module HeapTree (HTree) where

import SigHeap

data Ord a => HTree a = Null
                      | Fork a (HTree a) (HTree a)

instance Heap HTree where

   empty = Null

   isEmpty Null = True
   isEmpty _    = False

   minElem (Fork x _ _) = x
```

```
deleteMin (Fork _ a b) = merge a b

insert x ht = merge (Fork x Null Null) ht

merge l Null = l
merge Null r = r
merge l@(Fork x a b) r@(Fork y c d)
 | x<y  = let (p,q) = reduce2 a b r
          in Fork x p q
 | x>=y = let (p,q) = reduce2 l c d
          in Fork y p q
 where reduce2 a b c = (a, merge b c)  -- (merge b c, a)
                       -- (b, merge a c)     (merge a c, b)
                       -- (c, merge a b)     (merge a b, c)
```

Sortieren durch Einfügen in einen Heap

Unterschied zu Original-Heapsort:
dort Herstellen der Heapeigenschaft in einer bereits vorhandenen Datenmenge

```
insertSort :: (Heap t, Ord a) => t a -> [a] -> [a]
insertSort nullTree xs = let heap = foldl (flip insert) nullTree xs
                         in fromHeap heap
```

Der Typ von `nullTree` wird benutzt, um `insertSort` die Heapimplementierung mitzuteilen, die benutzt werden soll.

```
*HeapTree> insertSort (empty :: HTree Int) [1,4,8,3,7,2,9,0,5,6]
[0,1,2,3,4,5,6,7,8,9]
```

Effizienzproblem

- `merge`-Funktion führt mergen dreier HeapTrees auf zwei HeapTrees zurück

```
reduce2 a b c = (a, merge b c)  -- (merge b c, a)
                -- (b, merge a c)     (merge a c, b)
                -- (c, merge a b)     (merge a b, c)
```

- Problem: es kann häufig der tiefste HeapTree beim rekursiven Merge involviert sein
- Lösung: Maxiphobic HeapTree (meidet größten Teilheap)
  - merken der Größe eines Heaps
  - mergen der beiden kleineren Heaps

```
module MPHeapTree (MPTree) where
 import SigHeap

data Ord a => MPTree a = Null
                       | Fork Int a (MPTree a) (MPTree a)

instance Heap MPTree where
   empty = Null

  isEmpty Null = True
  isEmpty (Fork _ _ _ _) = False

  minElem (Fork _ x _ _) = x

  deleteMin (Fork _ _ a b) = merge a b

  insert x ht = merge (Fork 1 x Null Null) ht
```

```
merge l Null = l
merge Null r = r
merge a b
 | minElem a <= minElem b = join a b
 | otherwise              = join b a
 where
  join (Fork n x a b) c = Fork (n + size c) x aa (merge bb cc) where

     (aa,bb,cc) = orderBySize a b c
     orderBySize a b c
      | size a == biggest = (a,b,c)
      | size b == biggest = (b,a,c)
      | size c == biggest = (c,a,b)
     biggest = size a `max` size b `max` size c
     size Null = 0
     size (Fork n _ _ _) = n
```

Vergleich

	5000	10000	20000
HeapTree	2.0	10.3	72.6
Maxiphobic HeapTree	0.1	0.4	1.1
`quickSortFast`	0.0	0.2	0.7

Feststellung:

die Strategie ist erfolgreich, aber noch nicht gut genug

Round-Robin HeapTree

- Vermeidung der Zählervariablen
- Mergen in dem Zweig, in dem das letzte Mal nicht gemerged wurde

```
module RRHeapTree (RRTree) where

import SigHeap

data Ord a => RRTree a = Null
                       | ForkA a (RRTree a) (RRTree a)
                       | ForkB a (RRTree a) (RRTree a)
```

```
instance Heap RRTree where

   empty = Null

   isEmpty Null = True
   isEmpty _    = False

   minElem (ForkA x _ _) = x
   minElem (ForkB x _ _) = x

   deleteMin (ForkA _ a b) = merge a b
   deleteMin (ForkB _ a b) = merge a b
```

```
insert x ht = merge (ForkA x Null Null) ht

merge l Null = l
merge Null r = r
merge a b
 | minElem a <= minElem b = join a b
 | otherwise              = join b a
 where
   join (ForkA x a b) c = ForkB x (merge a c) b
   join (ForkB x a b) c = ForkA x a (merge b c)
```

Skew HeapTrees

Vermeidung zweier `Fork`-Konstruktoren durch Vertauschen der Zweige beim Einfügen

```
module SkewHeapTree (SkewTree) where

import SigHeap

data Ord a => SkewTree a = Null
                         | Fork a (SkewTree a) (SkewTree a)
```

```
instance Heap SkewTree where
   empty = Null

   isEmpty Null = True
   isEmpty _    = False

   minElem (Fork x _ _) = x

   deleteMin (Fork _ a b) = merge a b

   insert x ht = merge (Fork x Null Null) ht

   merge l Null = l
   merge Null r = r
   merge a b | minElem a <= minElem b  = join a b
             | otherwise               = join b a
    where join (Fork x a b) c = Fork x b (merge a c)
```

Laufzeitmessungen

```
module Main where

import SigHeap
import HeapTree
import MPHeapTree
import RRHeapTree
import SkewHeapTree
import QuicksortST
import List(partition,sort)
import CPUTime

checkForce :: Ord a => [a] -> Bool
checkForce [] = True
checkForce [x] = True
checkForce (x:y:r) = x<=y && checkForce (y:r)
```

```
insertSort :: (Heap t, Ord a) => t a -> [a] -> [a]
insertSort nullTree xs = let heap = foldl (flip insert) nullTree xs
                         in fromHeap heap

mergeSort :: Ord a => [a] -> [a]
mergeSort xs = mergeSort' (length xs) xs where
  mergeSort' n xs =
    if n<2 then xs
           else let n' = n`div` 2
                    (ys,zs) = splitAt n' xs
                in merge (mergeSort' n' ys) (mergeSort' (n-n') zs)
      where merge (x:xs) (y:ys) = if x<=y then x : merge xs (y:ys)
                                          else y : merge (x:xs) ys
            merge [] ys = ys
            merge xs [] = xs
```

```
singleTest :: Int -> ([Int] -> [Int]) -> IO String
singleTest n sf
 = do
    let xs = [ i^2 `mod` (n`div`10) | i<-[0..n-1] ]::[Int]
    if (xs==xs) then do
                      start <- getCPUTime
                      let ys = sf xs
                      if checkForce ys
                         then do
                                end <- getCPUTime
                                let y = show ((end-start)
                                              `div`100000000000)
                                return (init y ++"."++[last y])
                         else return ("not sorted!")
                else return ("?")
```

```
type Algo = (String,[Int] -> [Int])

seriesTest :: Algo -> [Int] -> IO ()
seriesTest (name,sf) ns
   = do times <- mapM (\n -> singleTest n sf) ns
        putStr (name ++ ": ")
        mapM_ (\ (n,t) -> putStr ("("++show n++":"++t++"),"))
              (zip ns times)
        putStrLn ""

htree, mphtree  ... :: Algo
htree   = ("HTree",   insertSort (empty::HTree  Int))
mphtree = ("MPHTree", insertSort (empty::MPTree Int))
main :: IO ()
main = do seriesTest htree    [3000,6000,12000]
          seriesTest mphtree  [3000,6000,12000]
```

Laufzeitvergleiche (1)

	50000	100000	200000
`quickSortFast`	3.3	11.3	42.5
MaxiPhobic Heap	3.0	8.5	24.0
Round-Robin Heap	0.8	2.3	6.5
Skew Heap	0.7	2.0	5.4
`mergeSort`	0.6	1.4	3.4
`List.sort`	0.4	1.0	2.5

Laufzeitvergleiche (2)

	250000	500000	1000000
Skew Heap	8.4	19.6	51.7
`mergeSort`	4.4	10.2	24.0
`List.sort`	3.7	8.8	20.9

Effizienz von Speicherverbrauch und Laufzeit

Einfluß der Reduktionsstrategie auf die Anzahl der Reduktionen

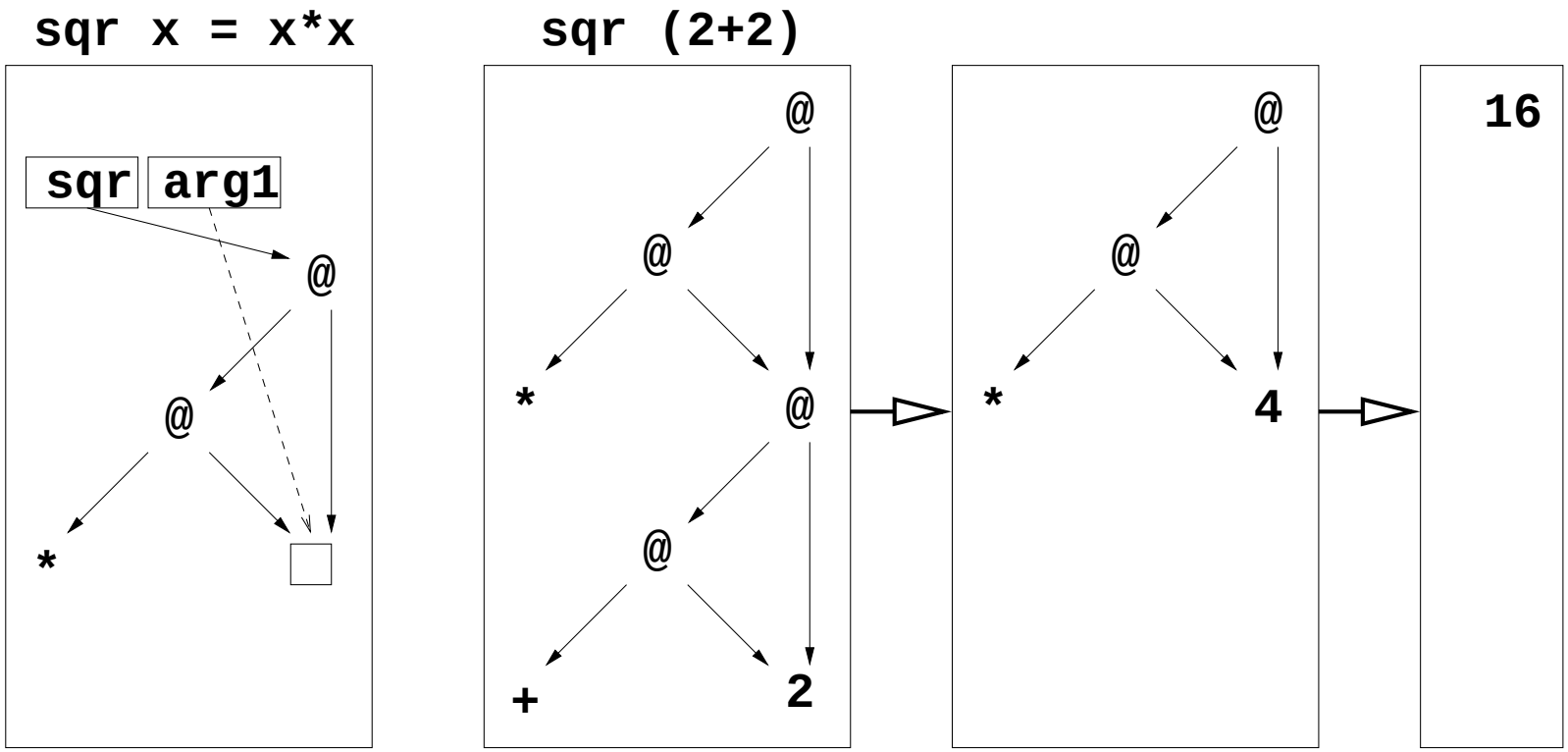
sei `sqr x = x*x`:

applicative order	normal order	applicative order	normal order
`sqr (3+4)`		`fst (sqr 4, sqr 2)`	
`sqr 7`	`(3+4)*(3+4)`	`fst (4*4, sqr 2)`	`sqr 4`
`7*7`	`7*(3+4)`	`fst (16, sqr 2)`	`4*4`
`49`	`7*7`	`fst (16, 2*2)`	`16`
	`49`	`fst (16, 4)`	
		`16`	

Sharing durch Graphreduktion

Graph-Reduktion erlaubt: normal-order Reduktion
ohne Datenduplikation oder Mehrfachauswertung

Beispiel:



Einfluß der Reduktionsstrategie auf die Schrittzahl

Ausdruck	`sqr (3+4)`	`fst (sqr 4, sqr 2)`
applicative-order	3	5
normal-order ohne Sharing	4	3
normal-order mit Sharing	3	3

Lazy-Auswertung (normal-order mit Sharing) braucht nie mehr Reduktionsschritte als eager-Auswertung (applicative-order), meistens erheblich weniger!

Metrik für Zeit- und Speicherbedarf

Seien E_i Ausdrücke, $E_0 \rightarrow E_1 \rightarrow ... \rightarrow E_n$ die Folge der (elementaren) Reduktionen in der Berechnung.

- Zeitbedarf: n Schritte
  Abstraktion von (u.U. erheblichen) Schwankungen der Realzeiten zwischen unterschiedlichen Reduktionen (z.B. bedingt durch Graphgrößen).
- Speicherbedarf: max { Graphgröße für $E_i \mid 0 \leq i \leq n$ }
  Abstraktion von
  - Space-Leaks (optimistisch)
  - Erzeuger/Verbraucher-Beziehungen (pessimistisch)

Unter *Space-Leaks* versteht man im Speicher residente Datenstrukturen, die unter keinen Umständen mehr gebraucht werden, aber (temporär) nicht als solche erkannt werden.

Weitere vereinfachende Annahmen

- eager-Auswertung
  - pessimistisch für die Anzahl der Reduktionen bei Laziness
  - optimistisch bzgl. des Speicherplatzverbrauchs und, in der Folge, der Konstanz des Zeitbedarfs pro Schritt
- konstante Zeit für bestimmte Operationen, z.B. für `*` in (`foldl (*) 1`), (gilt für `Int`, aber nicht für `Integer`)
- für Elemente unbeschränkter Datenstrukturen (z.B. Listen, Bäume, Arrays)
  1. Speicherbedarf konstant
  2. alle Operationen in konstanter Zeit durchführbar

Asymptotische Analyse

Sei g eine (partielle) Funktion vom Typ $T = \mathbb{N}^a \to \mathbb{N}$.

- $O(g) \stackrel{\text{def}}{=} \{\ f \in T \mid \exists k \in \mathbb{N} : (\forall^* x \in \text{dom}(g) : f(x) \leq k \cdot g(x))\ \}$
- $\Omega(g) \stackrel{\text{def}}{=} \{\ f \in T \mid \exists k \in \mathbb{N} : (\forall^* x \in \text{dom}(g) : f(x) \geq k^{-1} \cdot g(x))\ \}$
- $\Theta(g) \stackrel{\text{def}}{=} O(g) \cap \Omega(g)$

wobei $\forall^* x : \varphi \stackrel{\text{def}}{=} (\exists n : n \in \mathbb{N} : (\forall x : (\forall i : 0 \leq i < a : x_i > n) : \varphi))$

Um Nebenbedingungen sprachlich formulieren zu können, werden die Funktionen nicht als λ-Ausdruck, sondern durch ihren Rumpf mit expliziter Nennung der Variablen angegeben.

Beispiel: "Mit p $(p > 0)$ Prozessoren kann man n $(n \geq p)$ boolesche Werte in der Zeit $O(n/p + \lceil \log_2 p \rceil)$ mit $\wedge$ verknüpfen."
(Eine brauchbare Analyse bedarf zweier Variablen: n und p.)

Rechnen mit der Θ-Notation

Abkürzende Schreibweise (Rechnen mit Repräsentanten der Klasse):
$\Theta(E)$ als Teil eines arithmetischen Ausdrucks bedeutet: (E', wobei $E' \in \Theta(E)$)

Achtung: $f = O(g)$ nicht anstelle von $f \in O(g)$ verwenden!
Problem: $1 \in O(1)$ und $1 \in O(n)$, aber $O(1) \neq O(n)$

Rechenregeln für Θ (gleichermaßen für O und Ω)*

- Linearität (g Ausdruck in Parametern, $a > 0$ und b konstant):
  $\Theta(a \cdot g + b) = \Theta(g)$
- Distributivität (f, g Ausdrücke in Parametern)
  - $\Theta(f) + \Theta(g) = \Theta(f + g)$
  - $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$

(*) Reischuk: Einführung in die Komplexitätstheorie, Teubner-Verlag

Zeitanalyse

Def.: $T[\![f]\!][n_1, ..., n_k]$ die Zeit für die Funktionsdefinition f bei einer Eingabe mit Grössenparametern n_1 bis n_k.

Achtung: `rev == rev1` (Semantik), aber $[\![\texttt{rev}]\!] \neq [\![\texttt{rev1}]\!]$ (Syntax)

Beispiele von Analyseergebnissen:

1. ```
   rev [] = []
   rev (x:xs) = rev xs ++ [x]
   ```
   $n = \texttt{length xs} : T[\![\texttt{rev}]\!][n] = \Theta(n^2)$
2. ```
   rev1 xs = foldl (flip (:)) [] xs
   ```
   $n = \texttt{length xs} : T[\![\texttt{rev1}]\!][n] = \Theta(n)$
3. ```
   xs ++ ys = foldr (:) ys xs
   ```
   $n = \texttt{length xs}, m = \texttt{length ys} : T[\![\texttt{(++)}]\!][n, m] = \Theta(n)$
4. ```
   concat xss = foldr (++) [] xss
   ```
   $m = \texttt{length xss}, (\forall i : 0 \leq i < m : n = \texttt{length (xss!!i)})$
   $T[\![\texttt{concat}]\!][m, n] = \Theta(m \cdot n)$

Zeitanalyse der Funktion `rev`

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

Abstrakte Interpretation: reduziere Listeninformation auf ihre Länge.

man erhält rekursive Zeitfunktion für `rev` mit $n =$ `length xs`:

$T[\![\texttt{rev}]\!][0] = \Theta(1)$
$T[\![\texttt{rev}]\!][n+1] = \Theta(1) + T[\![\texttt{rev}]\!][n] + T[\![\texttt{(++)}]\!][n, 1]$

Lösung der Rekursion:

$T[\![\texttt{rev}]\!][n] = \Theta(1) + \sum_{i=0}^{n-1}(\Theta(1) + T[\![\texttt{(++)}]\!][i, 1])$

Es gilt: $T[\![\texttt{(++)}]\!][i, 1] = \Theta(i)$, also $T[\![\texttt{rev}]\!][n] = \sum_{i=0}^{n-1} \Theta(i) = \Theta(n^2)$

Flattening eines Binärbaums

```
data BTree a = Leaf a | Fork (BTree a) (BTree a)
flatten :: BTree a -> [a]
flatten (Leaf x)      = [x]
flatten (Fork xt yt) = flatten xt ++ flatten yt
```

Zeitfunktion für einen vollständigen Baum der Höhe h:

$$\begin{aligned} T[\![\texttt{flatten}]\!][0] &= \Theta(1) \\ T[\![\texttt{flatten}]\!][h+1] &= 2\,T[\![\texttt{flatten}]\!][h] + T[\![\texttt{(++)}]\!][2^h, 2^h] \\ &= 2\,T[\![\texttt{flatten}]\!][h] + \Theta(2^h) \end{aligned}$$

Lösung: $T[\![\texttt{flatten}]\!][h] = \Theta(h \cdot 2^h)$

Anzahl der Schritte: $\Theta(s \cdot \log s)$ für einen Baum mit s Knoten.

Verbesserung durch einen Akkumulator

Anforderung:

```
flatcat :: BTree a -> [a] -> [a]
flatcat xt xs == flatten xt ++ xs
```

Implementierung:

```
flatcat (Leaf x)     xs = x:xs
flatcat (Fork xt yt) xs = flatcat xt (flatcat yt xs)
```

Zeitanalyse für einen Baum der Höhe h, $n =$ `length xs`:
$T[\![\texttt{flatcat}]\!][0, n] = \Theta(1)$
$T[\![\texttt{flatcat}]\!][h+1, n] = \Theta(1) + T[\![\texttt{flatcat}]\!][h, 2^h + n] + T[\![\texttt{flatcat}]\!][h, n]$

Lösung: $T[\![\texttt{flatcat}]\!][h, n] = \Theta(2^h)$

Anzahl der Schritte: $\Theta(s)$ für einen Baum mit s Knoten.

Tupling: Generalisierung des Ergebnisses

(Akkumulator-Technik: Generalisierung des Arguments)

Bsp.: Fibonacci-Funktion

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

Zeitanalyse (n der Wert des Arguments von `fib`):

$T[\![\texttt{fib}]\!][0] = \Theta(1)$
$T[\![\texttt{fib}]\!][1] = \Theta(1)$
$T[\![\texttt{fib}]\!][n+2] = T[\![\texttt{fib}]\!][n] + T[\![\texttt{fib}]\!][n+1] + \Theta(1)$

Lösung: $T[\![\texttt{fib}]\!][n] = \Theta(\phi^n)$, wobei $\phi = (1+\sqrt{5})/2 \approx 1.62$

Anforderung:

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

Implementierung:

```
fibtwo 0       = (0,1)
fibtwo n | n>0 = let (a,b) = fibtwo (n-1)
                 in (b,a+b)
```

Verwendung:

```
fib n = fst (fibtwo n)
```

Zeitbedarf: $\Theta(n)$

Amortisierende Kosten, Bsp. Warteschlange

- Ausgelesen wird an dem gegenüberliegenden Ende des Einfügens
- bei Verwendung einer einzelnen Liste
  - entweder Einfügen oder Auslesen eines Elements verursacht Kosten $\Theta(n)$ bei Schlangenlänge n.
  - Gesamtkosten $(*)$ für vollständiges Einfügen und dann vollständiges Auslesen von n Elementen: $\sum_{i=0}^{n-1} \Theta(1) + \sum_{i=0}^{n-1} \Theta(n) = \Theta(n^2)$
- Verwendung zweier Listen
  - in Eingangsliste wird vorne eingefügt
  - aus Ausgangsliste wird vorne entfernt
  - ist Ausgangsliste leer, wird Eingangsliste revertiert zur Ausgangsliste
  - Gesamtkosten analog zu $(*)$: $\Theta(n)$
    1. einfügen/auslesen: $\sum_{i=0}^{n-1} \Theta(1) = \Theta(n)$
    2. revertieren: $\Theta(n)$

```
module Queue (Queue,initQueue,isEmpty,enqueue,dequeue) where

data Queue a = Q { inQ::[a], outQ::[a] }

initQueue :: Queue a
initQueue = Q { inQ=[], outQ=[] }

isEmpty :: Queue a -> Bool
isEmpty q = null (inQ q) && null (outQ q)

enqueue :: Queue a -> a -> Queue a
enqueue q x = let inQ' = inQ q
              in q { inQ = x : inQ' }
```

```
dequeue :: Queue a -> Maybe (a, Queue a)
dequeue q = let (r:rs) = reverse (inQ q)
                (o:os) = outQ q
            in if isEmpty q
                  then Nothing
                  else Just (if null (outQ q)
                                then (r, Q {inQ=[], outQ=rs})
                                else (o, q {outQ=os}))
```

- das Auslesen eines bestimmten Elements kann $\Theta(n)$ Operationen erfordern (zum Revertieren)
- Amortisation: dabei nehmen auch $\Theta(n)$ andere Elemente am Revertieren teil
- jedes in der Schlange befindliche Element nimmt nur einmal an einer Revertierung teil, verursacht also insgesamt einen konstanten Kostenbeitrag an den Schlangenoperationen

Speicherverbrauch

Bsp.: Durchschnittsberechnung

```
average xs = sum xs / length xs

average' xs = su / len
 where (su,len) = foldl ( (x,y) e -> (x+e,y+1) ) (0,0) xs
```

Sei n die Länge von `xs`:

- Beide Varianten brauchen Zeit $\Theta(n)$
- Unterschiede im Speicherverbrauch bei Anwendung auf `[1..n]`
  - `average`: Speicherbedarf $\Theta(n)$,
    `[1..n]` wird vollständig erzeugt, weil zweimal gebraucht
  - `average'`: Speicherbedarf $\Theta(1)$,
    jedes erzeugte Element kann sofort nach Gebrauch wieder freigegeben werden;
    Voraussetzung: Reduktionsoperator wird strikt angewendet

Nicht-strikte `foldl` vs. strikte `sfoldl`-Anwendung

```
sfoldl f e []     = e
sfoldl f e (x:xs) = let y = f e x
                    in  y `seq` sfoldl f y xs
```

```
   foldl (+) 0 [1..4]
→  foldl (+) (0+1) [2..4]
→  foldl (+) ((0+1)+2) [3..4]
→  foldl (+) (((0+1)+2)+3) [4]
→  foldl (+) ((((0+1)+2)+3)+4) []
→  (((0+1)+2)+3)+4
→  ((1+2)+3)+4
→  (3+3)+4
→  6+4
→  10
```

```
   sfoldl (+) 0     [1..4]
→  sfoldl (+) (0+1) [2..4]
→  sfoldl (+) 1     [2..4]
→  sfoldl (+) (1+2) [3..4]
→  sfoldl (+) 3     [3..4]
→  sfoldl (+) (3+3) [4]
→  sfoldl (+) 6     [4]
→  sfoldl (+) (6+4) []
→  sfoldl (+) 10    []
→  10
```

Strikte Anwendung: konstanter Speicherbedarf

Space Leaks, Bsp.: Funktionales Quicksort

```
qsort [] = []
qsort (x:xs) = let p = partition (<x) xs
               in qsort (fst p) ++ [x] ++ qsort (snd p)
```

- Annahme: xs ist absteigend vorsortiert
- (`fst p`) immer gleich `xs`, (`snd p`) immer leer
- Es werden alle Postfixe (wegen `partition` ohne Sharing) gleichzeitig im Speicher gehalten: $\Omega(n^2)$ Speicherbedarf
- Aufruf `qsort (fst p)`: Freigabe von `p` nach dem Match von (`x:xs`) gegen (`fst p`) nicht möglich!
  Grund: (`snd p`) hält Referenz auf `p`.
- Abhilfemöglichkeit: Fusion von `qsort` und `partition` [Bird-Buch]
  $\Rightarrow$ Speicher $O(n)$