Funktionales Programmieren
(Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

Vorlesung vom 19.01.2004: Effiziente Funktionale Programme

Inhalt

- Nachschlag: Verbunde (labelled records) in Haskell
- Effizienz
  - Zeitbedarf: Endrekursion — `while` in Haskell
  - Platzbedarf: Speicherlecks
  - Weitere Performanz-Fallen
- Zusammenfassung: Funktionales Programmieren
  - Stärken und Schwächen
  - Andere Funktionale Programmiersprachenen

Verbunde (Labelled Records)

Probleme mit großen Datentypen

- Beispiel Warenverwaltung
  - Ware mit Bezeichung, Stückzahl, Preis (in Cent)

```
data Item = Item String Int Int
```

  - Kommt Stückzahl oder Preis zuerst?

Probleme mit großen Datentypen

- Beispiel Warenverwaltung
  - Ware mit Bezeichung, Stückzahl, Preis (in Cent)

```
data Item = Item String Int Int
```

  - Kommt Stückzahl oder Preis zuerst?

- Beispiel Buch:
  - Titel, Autor, Verlag, Signatur

```
data Book' = Book' String String String String
```

  - Kommt Titel oder Autor zuerst?
    Ist der Verlag das dritte oder vierte Argument?

- Reihenfolge der Felder.
  - Typsynonyme (`type Author= String`) helfen nicht
  - Neue Typen (`data Author = Author String`) sind umständlich
- Zugriff und "Überschreiben"
  - Für jedes Feld einzeln zu definieren.

```
getSign :: Book'-> String
getSign (Book' _ _ _ s) = s
setSign :: Book'-> String-> Book'
setSign (Book' t a p _) s = Book' t a p s
```

- Inflexibilität
  - Wenn neues Feld hinzugefügt wird, alle Konstruktoren ändern.

Lösung: *labelled records*

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

```
data Book = Book { author :: String,
                   title  :: String,
                   publisher :: String }
```

- Konstruktion:

```
b = Book
  {author = "M. Proust",
   title  = "A la recherche du temps perdu",
   publisher = "S. Fischer Verlag"}
```

- Selektion durch Feldnamen:

```
publisher b --> "S. Fischer Verlag"
author b    --> "M. Proust"
```

- Update:

```
b{publisher = "Rowohlt Verlag"}
```

  - Rein funktional! (`b` bleibt unverändert)

- Patternmatching:

```
print :: Book-> IO ()
print (Book{author= a, publisher= p, title= t}) =
  putStrLn (a++ " schrieb "++ t ++ " und "++
            p++ " veröffentlichte es.")
```

- Partielle Konstruktion und Patternmatching möglich:

```
b2 = Book {author= "C. Lüth"}
shortPrint :: Book-> IO ()
shortPrint (Book{title= t, author= a}) =
  putStrLn (a++ " schrieb "++ t)
```

- Verbunde sind leicht erweiterbar:

```
data Book = Book {author :: String,
                  title  :: String,
                  publisher :: String,
                  signature :: String }
```

  Programm muß nicht geändert werden (nur neu übersetzt).

Effizienz

Endrekursion

Eine Funktion ist endrekursiv, wenn kein rekursiver Aufruf in einem geschachtelten Ausdruck steht.

- D.h. darüber nur `if`, `case`, guards oder Fallunterscheidungen.
- Entspricht `while` in imperativen Sprachen.
- Wird als Schleife implementiert.
- Nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` ist 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 n acc = if n == 0 then acc
                          else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` nicht. (Zeigen)

- Liste umdrehen, nicht endrekursiv:

```
rev' :: [a]-> [a]
rev' []     = []
rev' (x:xs) = rev' xs ++ [x]
```

  - Hängt auch noch hinten an — $O(n^2)$!

- 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 in $O(n)$: (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': 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

```
length' :: [a]-> Int
length' xs = if (null xs) then 0
             else 1+ length' (tail xs)
```

- Zuordnung der Variablen:

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

- 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 0 + y
             else len (tail xs) (1+ y)
```

- Allgemeines Muster:
  - Monoid (ϕ, e): ϕ assoziativ, e neutrales Element.
  - Zustäzlicher Parameter akkumuliert das Resultat.
    (Fast wie eine Zustandsvariable!)

Endrekursive Aktionen (zum Nachlesen!)

Eine Aktion ist endrekursiv, wenn nach dem rekursiven Aufruf keine weiteren Aktionen folgen.

- 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 instantiierte Funktion
- Beispiel: die Klasse *Show*
  - Nur `show` wäre zu langsam ($O(n^2)$):

```
class Show a where
   show :: a-> String
```

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch closures
  - closure: partiell instantiierte Funktion
- Beispiel: die Klasse *Show*
  - Nur `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 ($O(n)$).

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

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

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen

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

- 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.
  - Nicht mehr benutzt: Bezeichner nicht mehr im Scope.
- Eine Funktion hat ein Speicherleck, wenn Speicher belegt wird, der nicht mehr benötigt wird.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden nicht auswertet.
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Striktheit durch erzwungene Auswertung:
  - `seq :: a-> b-> b` wertet erstes Argument aus.
  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung

    ```
    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.
- `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` endrekursiv, traversiert aber immer die ganze Liste.
- `foldl'` konstanter Platzaufwand, traversiert aber auch die ganze Liste.
- Wann welches `fold`?
  - Strikte Funktionen mit `foldl'` falten.
  - Wenn nicht die ganze Liste benötigt wird, `foldr`:

```
all :: (a-> Bool)-> [a]-> Bool
all p = foldr ((&&) . p) True
```

Gemeinsame Teilausdrücke

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

```
f :: Int-> Int
f x = (x+ 1)* (x+ 1)
```

```
f (trace "Foo" (3+2))
x + x where x = (trace "Bar" (3+2))
```

- *Sharing* von Teilausdrücken (wie `x`)
  - Explizit mit `where` oder `let` oder
  - Implizit (Optimierung in `ghc`)

Memoisation

- Kleine Änderung der Fibonacci-Zahlen als Strom:

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

- Große Wirkung: Exponentiell in Space/Time. Warum?

Memoisation

- Kleine Änderung der Fibonacci-Zahlen als Strom:

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

- Große Wirkung: Exponentiell in Space/Time. Warum?
  - Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.
- Memoisation: Bereits berechnete Ergebnisse speichern.
- In Hugs: Aus dem Modul `Memo`:

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

- Damit ist alles wieder gut (oder?)

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

Weitere Performanz-Fallen

Überladene Funktionen

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: dictionaries von Klassenfunktionen.
  - Überladen wird zur Laufzeit aufgelöst (dynamisches Binbden)
- 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:
  - Festgelegte Länge
  - Zugriff auf n-tes Element in konstanter Zeit.
  - Abstrakt: Abbildung eines Index auf Daten

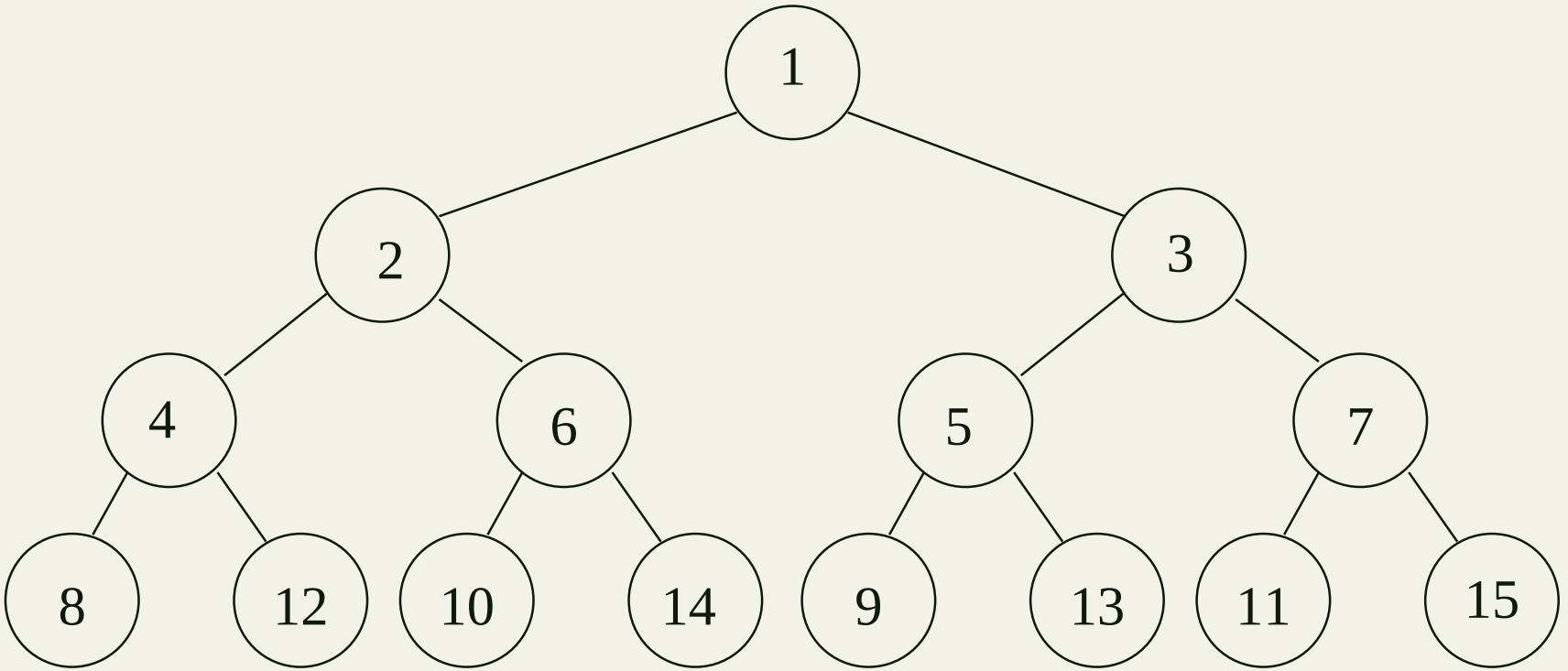
- Modul `Array` aus der Standardbibliothek

```
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.
- In Hugs/GHC vordefiniert (als "primitiver" Datentyp)

Funktionale Arrays

- Idee: Arrays implementiert durch binäre Bäume
- Pfad im Index kodiert: 0 — links, 1 — rechts:



- Schnittstelle:

```
module FArray(
  Array, -- abstract
  (!),  -- :: Array a-> Int-> Maybe a,
  upd,  -- :: Array a-> Int-> a-> Array a,
  remv, -- :: Array a-> Int-> Array a
  ) where
```

- Der Basisdatentyp ist ein binärer Baum:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
                   deriving (Show, Read)
type Array a = Tree a
```

  - Invariante: zusammenhängende Indizes.

- Lookup: Baumtraversion

```
(!) :: Tree a -> Int-> Maybe a
Leaf ! _                = Nothing
(Node v t1 t2) ! k
    | k == 1     = Just v
    | even k     = t1 ! (k `div` 2)
    | otherwise = t2 ! (k `div` 2)
```

- Update: ähnlich.
  - Feld kann erweitert werden, aber immer nur um ein Element!

```
upd :: Tree a -> Int-> a-> Tree a
upd Leaf k v =
  if k== 1 then (Node v Leaf Leaf)
  else error "Tree.upd: illegal index"
upd (Node w t1 t2) k v
  | k == 1     = Node v t1 t2
  | even k     = Node w (upd t1 (k `div` 2) v) t2
  | otherwise = Node w t1 (upd t2 (k `div` 2) v)
```

- Remove: darf nur für oberstes Element angewandt werden.

```
remv :: Tree a-> Int-> Tree a
remv Leaf _  = Leaf
remv (Node w t1 t2) k
   | k == 1     = Leaf
   | even k     = Node w (remv t1 (k `div` 2)) t2
   | otherwise = Node w t1 (remv t2 (k `div` 2))
```

- Mangel: Invariante wird bei `remv` nicht geprüft.
- Einfache Abhilfe:

```
type Array a = (Tree a, Int)
```

Funktionales Programmieren – Zusammenfassung und Ausblick

Wesentliche Konzepte

Wesentliche Konzepte

- Daten: rekursive parameterisierte Datenstrukturen (`data`)

Wesentliche Konzepte

- Daten: rekursive parameterisierte Datenstrukturen (`data`)
- Algorithmen: Funktionen (ohne Seiteneffekte)

Wesentliche Konzepte

- Daten: rekursive parameterisierte Datenstrukturen (`data`)
- Algorithmen: Funktionen (ohne Seiteneffekte)
- Kontrollstrukturen: Fallunterscheidung und Rekursion

Wesentliche Konzepte

- Daten: rekursive parameterisierte Datenstrukturen (`data`)
- Algorithmen: Funktionen (ohne Seiteneffekte)
- Kontrollstrukturen: Fallunterscheidung und Rekursion
- Typisierung: statisch, polymorph (in Haskell)

Wesentliche Konzepte

- Daten: rekursive parameterisierte Datenstrukturen (`data`)
- Algorithmen: Funktionen (ohne Seiteneffekte)
- Kontrollstrukturen: Fallunterscheidung und Rekursion
- Typisierung: statisch, polymorph (in Haskell)
- Funktionen höherer Ordnung

Fortgeschrittene Konzepte

Fortgeschrittene Konzepte

- Überladene Funktionen durch Typklassen

Fortgeschrittene Konzepte

- Überladene Funktionen durch Typklassen
- Unendliche Datenstrukturen und verzögerte Auswertung

Fortgeschrittene Konzepte

- Überladene Funktionen durch Typklassen
- Unendliche Datenstrukturen und verzögerte Auswertung
- Die reale Welt ist gekapselt (Monaden in Haskell)

Stärken

Stärken

- einfache, allgemeine Konstruktion von Daten
- kurze, intuitive Definition von Funktionen
  - syntaktischer Zucker: patterns, guards, Listenumschreibungen
- selbst definierbare Kontrollstrukturen (Kombinatoren)
- sichere und komfortable Typisierung (Inferenz)
- leichte Analyse: Korrektheit, Termination, Komplexität
- sicher gekapselte Aktionen (`do`, `IO a`)

Schwächen

Schwächen

- mangelnde Effizienz
- ungewohnte Schreibweisen
- mangelnde Eignung für die reale Softwareentwicklung

Schwächen

- mangelnde Effizienz
- ungewohnte Schreibweisen
- mangelnde Eignung für die reale Softwareentwicklung

Schwächen von Haskell:

- Komplexität der Sprache
- Dokumentation der Bibliotheken
  - z.B. im Vergleich zu Java's APIs

Schwächen

- mangelnde Effizienz
- ungewohnte Schreibweisen
- mangelnde Eignung für die reale Softwareentwicklung

Schwächen von Haskell:

- Komplexität der Sprache
- Dokumentation der Bibliotheken
  - z.B. im Vergleich zu Java's APIs

Aber: Stechen diese Argumente wirklich?

Reale Anwendungen in Haskell

- Spiele, Robotik, Musik (Hudak)
- GUI erzeugen mit `Haskell-Tk` (Bremen)
- Repositories verwalten mit UnForM-Workbench (Bremen)
- Web-Anwendungen programmieren (Lüth, Bremen)

Andere Funktionale Sprachen

Andere Funktionale Sprachen

- FP(Backus, 1978)
  - “rein”, strikt, dynamisch typisiert, Höherer Ordnung

Andere Funktionale Sprachen

- FP(Backus, 1978)
  - “rein”, strikt, dynamisch typisiert, Höherer Ordnung
- Standard ML (Milner, Gordon, 1979) Isabelle
  - “unrein”, strikt, statisch typisiert, Höherer Ordnung

Andere Funktionale Sprachen

- FP(Backus, 1978)
  - “rein”, strikt, dynamisch typisiert, Höherer Ordnung
- Standard ML (Milner, Gordon, 1979) Isabelle
  - “unrein”, strikt, statisch typisiert, Höherer Ordnung
- Erlang(Armstrong 1993) Telekommunikation
  - “unrein”, strikt, dynamisch typisiert, Erster Ordnung

Andere Funktionale Sprachen

- FP(Backus, 1978)
  - “rein”, strikt, dynamisch typisiert, Höherer Ordnung
- Standard ML (Milner, Gordon, 1979) Isabelle
  - “unrein”, strikt, statisch typisiert, Höherer Ordnung
- Erlang(Armstrong 1993) Telekommunikation
  - “unrein”, strikt, dynamisch typisiert, Erster Ordnung
- Clean (Plasmeijer, 1988) Simulation
  - ”rein”, verzögert, dynamisch typisiert, Höherer Ordnung
- OPAL (Pepper, 1987)

- rein, strikt, statisch typisiert, Höherer Ordnung

Zusammenfassung

- Verbunde in Haskell.
- Effizienz
  - Endrekursion: `while` für Haskell.
  - Speicherlecks
  - Überladene Funktionen
  - Arrays
- Schlussbemerkungen und Ausblick