

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 vom 20.11.2012: Funktionen Höherer Ordnung II

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
 - ▶ Einführung
 - ▶ Funktionen und Datentypen
 - ▶ Rekursive Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Funktionen höherer Ordnung II
 - ▶ Typinferenz
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Funktionen höherer Ordnung

- ▶ map und filter sind durch foldr darstellbar:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\beta$ ]  
map f = foldr ( $\lambda a b \rightarrow f a : b$ ) []
```

```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]  
filter p = foldr ( $\lambda a as \rightarrow$  if p a then a : as  
                    else as) []
```

Funktionen höherer Ordnung

- ▶ map und filter sind durch foldr darstellbar:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\beta$ ]  
map f = foldr ( $\lambda a b \rightarrow f a : b$ ) []
```

```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]  
filter p = foldr ( $\lambda a as \rightarrow$  if p a then a : as  
                    else as) []
```

foldr ist die **kanonische einfach rekursive** Funktion.

- ▶ Alle einfach rekursiven Funktionen sind als Instanz von foldr darstellbar.

$$\text{foldr } (:) [] = \text{id}$$

map als struktur erhaltenen Abbildung

map ist die kanonische **struktur erhaltende Abbildung**.

- ▶ **Struktur** (Shape) eines Datentyps T ist T (α).
- ▶ Für Listen: $[] \cong \text{Nat}$.
- ▶ Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length } (\text{map } f \text{ xs}) = \text{length } \text{xs}$$

Grenzen von foldr

- ▶ Andere rekursive Struktur über Listen
 - ▶ Quicksort: baumartige Rekursion

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort xs = qsort (filter (< head xs) xs) ++
           filter (head xs ==) xs ++
           qsort (filter (head xs <) xs)
```

- ▶ Rekursion nicht über Listenstruktur:
 - ▶ take: Rekursion über Int

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

- ▶ Version mit foldr divergiert für nicht-endliche Listen

fold für andere Datentypen

fold ist universell

Jeder algebraische Datentyp T hat genau ein foldr.

- ▶ Kanonische Signatur für T :
 - ▶ Pro Konstruktor C ein Funktionsargument f_C
 - ▶ Freie Typvariable β für T
- ▶ Kanonische Definition:
 - ▶ Pro Konstruktor C eine Gleichung
 - ▶ Gleichung wendet Funktionsparameter f_C auf Argumente an

```
data IL = Cons Int IL | Err String | Mt
```

```
foldIL :: (Int  $\rightarrow$   $\beta \rightarrow \beta$ )  $\rightarrow$  (String  $\rightarrow$   $\beta$ )  $\rightarrow$   $\beta \rightarrow$  IL  $\rightarrow$   $\beta$   
foldIL f e a (Cons i il) = f i (foldIL f e a il)  
foldIL f e a (Err str)   = e str  
foldIL f e a Mt          = a
```

fold für bekannte Datentypen

- ▶ Bool:

fold für bekannte Datentypen

- ▶ Bool: Fallunterscheidung:

```
data Bool = True | False
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 True = a1
```

```
foldBool a1 a2 False = a2
```

- ▶ Maybe a:

fold für bekannte Datentypen

- ▶ Bool: Fallunterscheidung:

```
data Bool = True | False
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 True = a1
```

```
foldBool a1 a2 False = a2
```

- ▶ Maybe a: Auswertung

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
```

```
foldMaybe ::  $\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe } \alpha \rightarrow \beta$ 
```

```
foldMaybe b f Nothing = b
```

```
foldMaybe b f (Just a) = f a
```

- ▶ Als maybe vordefiniert

fold für bekannte Datentypen

- ▶ Tupel:

fold für bekannte Datentypen

- ▶ Tupel: die uncurry-Funktion

```
foldPair :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha, \beta$ )  $\rightarrow$   $\gamma$   
foldPair f (a, b) = f a b
```

- ▶ Natürliche Zahlen:

fold für bekannte Datentypen

- ▶ Tupel: die uncurry-Funktion

```
foldPair :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha, \beta$ )  $\rightarrow$   $\gamma$   
foldPair f (a, b) = f a b
```

- ▶ Natürliche Zahlen: Iterator

```
type Nat = Int  -- data Nat = Zero | Succ Nat  
foldNat ::  $\beta \rightarrow (\beta \rightarrow \beta) \rightarrow \text{Nat} \rightarrow \beta$   
foldNat e f x | x == 0 = e  
foldNat e f x | x > 0 = f (foldNat e f (x-1))
```

fold für binäre Bäume

- ▶ Binäre Bäume:

```
data Tree  $\alpha$  = Mt | Node  $\alpha$  (Tree  $\alpha$ ) (Tree  $\alpha$ )
```

- ▶ Label **nur** in den Knoten

- ▶ Instanzen von Map und Fold:

```
mapT :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  Tree  $\alpha \rightarrow$  Tree  $\beta$ 
```

```
mapT f Mt = Mt
```

```
mapT f (Node a l r) =
```

```
  Node (f a) (mapT f l) (mapT f r)
```

```
foldT :: ( $\alpha \rightarrow \beta \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow$  Tree  $\alpha \rightarrow \beta$ 
```

```
foldT f e Mt = e
```

```
foldT f e (Node a l r) =
```

```
  f a (foldT f e l) (foldT f e r)
```

- ▶ Kein (offensichtliches) Filter

Funktionen mit fold und map

- ▶ Höhe des Baumes berechnen:

```
height :: Tree  $\alpha$   $\rightarrow$  Int
height = foldT ( $\lambda$ _ l r  $\rightarrow$  1 + l + r) 0
```

- ▶ Inorder-Traversierung der Knoten:

```
inorder :: Tree  $\alpha$   $\rightarrow$  [ $\alpha$ ]
inorder = foldT ( $\lambda$ a l r  $\rightarrow$  l ++ [a] ++ r) []
```

Kanonische Eigenschaften von foldT und mapT

- ▶ Auch hier gilt:

$$\text{foldTree Node Mt} = \text{id}$$

$$\text{mapTree id} = \text{id}$$

$$\text{mapTree } f \circ \text{mapTree } g = \text{mapTree } (f \circ g)$$

$$\text{shape } (\text{mapTree } f \text{ xs}) = \text{shape } \text{xs}$$

- ▶ Mit $\text{shape} :: \text{Tree } \alpha \rightarrow \text{Tree } ()$

Das Labyrinth

- ▶ Das Labyrinth als variadischer Baum:

```
data Lab a = Node a [Lab a]
```

- ▶ Auch hierfür `foldT` und `mapT`:

Das Labyrinth

- ▶ Das Labyrinth als variadischer Baum:

```
data Lab a = Node a [Lab a]
```

- ▶ Auch hierfür foldT und mapT:

```
foldT :: (a → [b] → b) → Lab a → b  
foldT f (Node a ns) = f a (map (foldT f) ns)
```

```
mapT :: (a → b) → Lab a → Lab b  
mapT f (Node a ns) = Node (f a) (map (mapT f) ns)
```

Suche im Labyrinth

► Tiefensuche via foldT

```
dfts' :: Lab a → [Path a]
dfts' n = foldT add n where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

Suche im Labyrinth

- ▶ Tiefensuche via foldT

```
dfts' :: Lab a → [Path a]
dfts' n = foldT add n where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

- ▶ Problem:

- ▶ foldT terminiert **nicht** für **zyklische** Strukturen
- ▶ Auch nicht, wenn add prüft ob a schon enthalten ist

Tiefensuche direkt

- ▶ Deshalb **direkte** rekursive Lösung

```
dfts :: Eq a => (Lab a -> Bool) -> Lab a -> [Path a]
dfts trg = dfts0 [] where
  dfts0 p n@(Node a ns)
    | trg n      = [reverse (a:p)]
    | elem a p  = []
    | otherwise = concatMap (dfts0 (a:p)) ns
```

- ▶ Funktioniert für **alle** Labyrinth
- ▶ Flexible Termination durch Prädikat
 - ▶ Blätter in gerichteten Graphen

```
isLeaf (Node a ns) = null ns
```

- ▶ Blätter in ungerichteten Graphen

```
isULeaf (Node a ns) = null ns || null (tail ns)
```

Zusammenfassung map und fold

- ▶ map und fold sind **kanonische** Funktionen höherer Ordnung
- ▶ Für jeden Datentyp definierbar
- ▶ foldl nur für Listen (**linearer** Datentyp)
- ▶ fold kann bei **zyklischen** Argumenten nicht terminieren
 - ▶ Problem: Termination von fold nur **lokal** entscheidbar

Funktionen Höherer Ordnung als Entwurfsmethodik

- ▶ **Kombination** von Basisoperationen zu komplexen Operationen
- ▶ **Kombinatoren** als **Muster** zur Problemlösung:
 - ▶ **Einfache** Basisoperationen
 - ▶ **Wenige** Kombinationsoperationen
 - ▶ Alle anderen Operationen **abgeleitet**
- ▶ **Kompositionalität**:
 - ▶ Gesamtproblem läßt sich **zerlegen**
 - ▶ Gesamtlösung durch **Zusammensetzen** der Einzellösungen

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K x y \triangleright x$$

$$S x y z \triangleright x z (y z)$$

$$I x \triangleright x$$

S , K , I sind **Kombinatoren**

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K x y \triangleright x$$

$$S x y z \triangleright x z (y z)$$

$$I x \triangleright x$$

S , K , I sind **Kombinatoren**

- ▶ Fun fact #1: kann alle berechenbaren Funktionen ausdrücken

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: **Kombinatorlogik** (Schönfinkel, 1924)

$$K \ x \ y \triangleright x$$

$$S \ x \ y \ z \triangleright x \ z \ (y \ z)$$

$$I \ x \triangleright x$$

S , K , I sind **Kombinatoren**

- ▶ Fun fact #1: kann alle berechenbaren Funktionen ausdrücken
- ▶ Fun fact #2: S und K sind genug: $I = S \ K \ K$

Beispiel: Parser

- ▶ **Parser** bilden Eingabe auf Parsierungen ab
 - ▶ Mehrere Parsierungen möglich
 - ▶ Backtracking möglich
- ▶ Kombinatoransatz:
 - ▶ **Basisparser** erkennen **Terminalsymbole**
 - ▶ **Parserkombinatoren** zur Konstruktion:
 - ▶ Sequenzierung (erst A , dann B)
 - ▶ Alternierung (entweder A oder B)
 - ▶ Abgeleitete Kombinatoren (z.B. Listen A^* , nicht-leere Listen A^+)

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse = ?

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = ?

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → b

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → (b, [a])

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → [(b, [a])]

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren
- ▶ Muss **mehrdeutige Ergebnisse** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

type Parse a b = [a] → [(b, [a])]

- ▶ Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- ▶ Parser übersetzt **Token** in **abstrakte Syntax**
- ▶ Muss **Rest der Eingabe** modellieren
- ▶ Muss **mehrdeutige Ergebnisse** modellieren
- ▶ Beispiel: "3+4*5" \rightsquigarrow [(3, "+4*5"),
(3+4, "*5"),
(3+4*5, "")]

Basisparser

- ▶ Erkennt **nichts**:

```
none :: Parse a b
none = const []
```

- ▶ Erkennt **alles**:

```
succeed :: b → Parse a b
succeed b inp = [(b, inp)]
```

- ▶ Erkennt **einzelne Token**:

```
spot :: (a → Bool) → Parse a a
spot p [] = []
spot p (x:xs) = if p x then [(x, xs)] else []
```

```
token :: Eq a ⇒ a → Parse a a
token t = spot (λc → t == c)
```

- ▶ Warum nicht none, succeed durch spot? Typ!

Basiskombinatoren: alt, >*>

- ▶ **Alternierung:**
 - ▶ Erste Alternative wird **bevorzugt**

```
infixl 3 'alt '  
alt  :: Parse a b → Parse a b → Parse a b  
alt p1 p2 i = p1 i ++ p2 i
```

Basiskombinatoren: alt, >*>

▶ Alternierung:

- ▶ Erste Alternative wird bevorzugt

```
infixl 3 'alt'  
alt :: Parse a b → Parse a b → Parse a b  
alt p1 p2 i = p1 i ++ p2 i
```

▶ Sequenzierung:

- ▶ Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>  
(>*>) :: Parse a b → Parse a c → Parse a (b, c)  
(>*>) p1 p2 i =  
  concatMap (\(b, r) →  
    map (\(c, s) → ((b, c), s)) (p2 r)) (p1 i)
```

Basiskombinatoren: use

- ▶ map für Parser (**Rückgabe** weiterverarbeiten):

```
infix 4 'use', 'use2'
```

```
use :: Parse a b → (b → c) → Parse a c
```

```
use p f i = map (λ(o, r) → (f o, r)) (p i)
```

```
use2 :: Parse a (b, c) → (b → c → d) → Parse a d
```

```
use2 p f = use p (uncurry f)
```

- ▶ Damit z.B. Sequenzierung **rechts/links**:

```
infixl 5 *>, >*
```

```
(*>) :: Parse a b → Parse a c → Parse a c
```

```
(>*) :: Parse a b → Parse a c → Parse a b
```

```
p1 *> p2 = p1 >*> p2 'use' snd
```

```
p1 >* p2 = p1 >*> p2 'use' fst
```

Abgeleitete Kombinatoren

- ▶ Listen: $A^* ::= AA^* \mid \epsilon$

```
list  :: Parse a b → Parse a [b]
list p = p >*> list p 'use2' (:)
        'alt' succeed []
```

- ▶ Nicht-leere Listen: $A^+ ::= AA^*$

```
some  :: Parse a b → Parse a [b]
some p = p >*> list p 'use2' (:)
        'alt' succeed []
```

- ▶ NB. Präzedenzen: $>*>$ (5) vor use (4) vor alt (3)

Verkapselung

▶ Hauptfunktion:

- ▶ Eingabe muß **vollständig** parsiert werden
- ▶ Auf **Mehrdeutigkeit** prüfen

```
parse :: Parse a b → [a] → Either String b
parse p i =
  case filter (null . snd) $ p i of
    []      → Left "Input does not parse"
    [(e, _)] → Right e
    _       → Left "Input is ambiguous"
```

▶ Schnittstelle:

- ▶ Nach außen nur Typ `Parse` sichtbar, plus **Operationen** darauf

Grammatik für Arithmetische Ausdrücke

$Expr ::= Term + Term \mid Term$

$Term ::= Factor * Factor \mid Factor$

$Factor ::= Variable \mid (Expr)$

$Variable ::= Char^+$

$Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$

Abstrakte Syntax für Arithmetische Ausdrücke

- ▶ Zur Grammatik **abstrakte Syntax**

```
data Expr    = Plus  Expr Expr
              | Times Expr Expr
              | Var   String
```

- ▶ Hier Unterscheidung Term, Factor, Number unnötig.

Parsierung Arithmetischer Ausdrücke

- ▶ Token: Char
- ▶ Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isAlpha) 'use' Var
         'alt' token '(' *> pExpr >* token ')'
```

- ▶ Parsierung von Term

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pFactor 'use2' Times
  'alt' pFactor
```

- ▶ Parsierung von Expr

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm 'use2' Plus
       'alt' pTerm
```

Die Hauptfunktion

- ▶ Lexing: Leerzeichen aus der Eingabe entfernen

```
parseExpr :: String → Expr
parseExpr i =
  case parse pExpr (filter (not.isSpace) i) of
    Right e → e
    Left err → error err
```

Ein kleiner Fehler

- ▶ **Mangel:** $a+b+c$ führt zu **Syntaxfehler** — Fehler in der **Grammatik**

- ▶ Behebung: **Änderung** der Grammatik

$$Expr ::= Term + Expr \mid Term$$
$$Term ::= Factor * Term \mid Factor$$
$$Factor ::= Variable \mid (Expr)$$
$$Variable ::= Char^+$$
$$Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$$

- ▶ **Abstrakte Syntax** bleibt

Änderung des Parsers

- ▶ Entsprechende Änderung des Parsers in pTerm

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pTerm 'use2' Times
  'alt' pFactor
```

- ▶ ... und in pExpr:

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr 'use2' Plus
  'alt' pTerm
```

- ▶ pFactor und Hauptfunktion bleiben.

Erweiterung zu einem Taschenrechner

- ▶ Zahlen:

$$\textit{Factor} ::= \textit{Variable} \mid \textit{Number} \mid \dots$$
$$\textit{Number} ::= \textit{Digit}^+$$
$$\textit{Digit} ::= 0 \mid \dots \mid 9$$

- ▶ Eine einfache **Eingabesprache**:

$$\textit{Input} ::= ! \textit{Variable} = \textit{Expr} \mid \$ \textit{Expr}$$

- ▶ Eine **Auswertungsfunktion**:

```
type State = [(String, Integer)]
```

```
eval :: State → Expr → Integer
```

```
run :: State → String → (State, String)
```

Zusammenfassung Parserkombinatoren

- ▶ **Systematische Konstruktion** des Parsers aus der Grammatik.
- ▶ **Kompositional:**
 - ▶ Lokale Änderung der Grammatik führt zu lokaler Änderung im Parser
 - ▶ Vgl. Parsergeneratoren (yacc/bison, antlr, happy)
- ▶ Struktur von Parse zur Benutzung irrelevant
 - ▶ Vorsicht bei **Mehrdeutigkeiten** in der Grammatik (Performance-Falle)
 - ▶ **Einfache Implementierung** (wie oben) skaliert **nicht**
 - ▶ Effiziente Implementation mit **gleicher Schnittstelle** auch für **große Eingaben** geeignet.

Zusammenfassung

- ▶ map und fold sind kanonische Funktionen höherer Ordnung
- ▶ ... und für alle Datentypen definierbar
- ▶ **Kombinatoren**: Funktionen höherer Ordnung als **Entwurfsmethodik**
 - ▶ Einfache **Basisoperationen**
 - ▶ Wenige aber **mächtige Kombinationsoperationen**
 - ▶ Reiche Bibliothek an **abgeleiteten** Operationen
- ▶ Nächste Woche: wie prüft man den Typ von

```
(>*>) p1 p2 i =  
  concatMap ( $\lambda(b, r) \rightarrow$   
    map ( $\lambda(c, s) \rightarrow ((b, c), s)$ ) (p2 r)) (p1 i)
```

↪ **Typinferenz!**