Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 vom 18.11.2014: Funktionen Höherer Ordnung II

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

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

Heute

- Die Geheimnisse von map und foldr gelüftet.
- map und foldr sind nicht nur für Listen.
- Funktionen höherer Ordnung als Entwurfsmuster

foldr ist kanonisch

- map und filter sind durch foldr darstellbar:

```
map :: (α→ β)→ [α]→ [β]
map f = foldr ((:). f) []
```

```
filter :: (α→ Bool)→ [α]→ [α]
filter p = foldr (λa as→ if p a then a:as
                                  else as) []
```

foldr ist kanonisch

- map und filter sind durch foldr darstellbar:

```
map :: (α → β) → [α] → [β]
map f = foldr ((:). f) []
```

```
filter :: (α → Bool) → [α] → [α]
filter p = foldr (λa as → 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 strukturerhalten Abbildung

map ist die kanonische strukturerhaltende Abbildung.

- Struktur (Shape) eines Datentyps $\mathsf{T}\ \alpha$ ist $\mathsf{T}\ ()$.
  - Für jeden Datentyp kann man kanonische Funktion $\mathsf{shape} :: \mathsf{T}\ \alpha \to \mathsf{T}\ ()$ angeben
  - Für Listen: $[()] \cong \mathsf{Nat}$.
- Für map gelten folgende Aussagen:

$$\mathsf{map\ id} = \mathsf{id}$$

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

$$\mathsf{shape}.\ \ \mathsf{map\ f} = \mathsf{shape}$$

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→ β→ β)→ (String→ β)→ β→ IL→ β
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 :: β→ β→ Bool→ β
foldBool a1 a2 True  = a1
foldBool a1 a2 False = a2
```

- Maybe a:

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = True | False

foldBool :: β→ β→ Bool→ β
foldBool a1 a2 True  = a1
foldBool a1 a2 False = a2
```

- Maybe a: Auswertung

```
data Maybe α = Nothing | Just α

foldMaybe :: β→ (α→ β)→ Maybe α→ β
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 :: (α→ β→ γ)→ (α, β)→ γ
foldPair f (a, b)= f a b
```

- Natürliche Zahlen:

fold für bekannte Datentypen

- Tupel: die uncurry-Funktion

```
foldPair :: (α → β → γ) → (α, β) → γ
foldPair f (a, b) = f a b
```

- Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat
foldNat :: β → (β → β) → Nat → β
foldNat e f Zero     = e
foldNat e f (Succ n) = f (foldNat e f n)
```

fold für binäre Bäume

- Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

  - Label nur in den Knoten

- Instanzen von Map und Fold:

```
mapT :: (α→ β)→ Tree α→ Tree β
mapT f Mt = Mt
mapT f (Node a l r)=
  Node (f a) (mapT f l) (mapT f r)
```

```
foldT :: (α→ β→ β→ β)→ β→ Tree α→ β
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 α→ Int
height = foldT (λ_ l r→ 1+ max l r) 0
```

- Inorder-Traversion der Knoten:

```
inorder :: Tree α→ [α]
inorder = foldT (λa l r→ l++ [a]++ r) []
```

Kanonische Eigenschaften von foldT und mapT

- Auch hier gilt:

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

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

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

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

- Mit shape :: Tree $\alpha \rightarrow$ Tree ()

Das Labyrinth

- Das Labyrinth als variadischer Baum:

  data VTree α = Node α [VTree α]

  type Lab α = VTree α

- Auch hierfür foldT und mapT:

Das Labyrinth

- Das Labyrinth als variadischer Baum:

```
data VTree α = Node α [VTree α]
```

```
type Lab α = VTree α
```

- Auch hierfür foldT und mapT:

```
foldT :: (α→ [β]→ β)→ VTree α→ β
foldT f (Node a ns) = f a (map (foldT f) ns)
```

```
mapT :: (α→ β)→ VTree α→ VTree β
mapT f (Node a ns) = Node (f a) (map (mapT f) ns)
```

Suche im Labyrinth

- Tiefensuche via foldT

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

Suche im Labyrinth

- Tiefensuche via foldT

```
dfts' :: Lab α→ [Path α]
dfts' = foldT add 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
  - Pfade werden vom Ende konstruiert

Alternativen: Breitensuche

- Alternative 1: Tiefensuche direkt rekursiv, mit Terminationsprädikat

```
dfts :: Eq α ⇒ (Lab α → Bool) → Lab α → [Path α]
```

- Alternative 2: Breitensuche für potentiell unendliche Liste aller Pfade

```
bfts :: Lab α → [Path α]
bfts l = bfts0 [] [l] where
  bfts0 p [] = []
  bfts0 p (Node a cs:ns) =
    reverse (a:p) : (bfts0 p ns ++ bfts0 (a:p) cs)
```

- Gegensatz zur Tiefensuche: Liste kann konsumiert werden

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
  - Im Labyrinth braucht man den Kontext um zu entscheiden ob ein Knoten ein Blatt ist

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)

$$
\begin{aligned}
K\ x\ y &\ \rhd\ x \\
S\ x\ y\ z &\ \rhd\ x\ z\ (y\ z) \\
I\ x &\ \rhd\ x
\end{aligned}
$$

S, K, I sind Kombinatoren

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein Kombinator ist ein punktfrei definierte Funktion höherer Ordnung.

- Herkunft: Kombinatorlogik (Schönfinkel, 1924)

$$
\begin{aligned}
K\ x\ y &\quad \triangleright \quad x \\
S\ x\ y\ z &\quad \triangleright \quad x\ z\ (y\ z) \\
I\ x &\quad \triangleright \quad x
\end{aligned}
$$

  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)

$$
\begin{aligned}
K\ x\ y &\ \triangleright\ x \\
S\ x\ y\ z &\ \triangleright\ x\ z\ (y\ z) \\
I\ x &\ \triangleright\ x
\end{aligned}
$$

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?

$$\textbf{type}\ \text{Parse} \ = ?$$

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = ?$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \mathsf{Parse}\ \alpha\ \beta = [\alpha] \to \beta$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \rightarrow (\beta, [\alpha])$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \to [(\beta, [\alpha])]$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \to [(\beta, [\alpha])]$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren
- Beispiel: `"4*5+3"` → [(4, `"*4+3"`),
  (4*5, `"+3"`),
  (4*5+3, `""`)]

Basisparser

- Erkennt nichts:

```
none :: Parse α β
none = const []
```

- Erkennt alles:

```
suceed :: β→ Parse α β
suceed b inp = [(b, inp)]
```

- Erkennt einzelne Token:

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

token :: Eq α⇒ α→ Parse α α
token t = spot (t ==)
```

  - Warum nicht none, suceed durch spot? Typ!

Basiskombinatoren: alt, $>*>$

- Alternierung:
  - Erste Alternative wird bevorzugt

```
infixl 3 `alt`
alt :: Parse α β → Parse α β → Parse α β
alt p1 p2 i = p1 i ++ p2 i
```

Basiskombinatoren: alt, >*>

- Alternierung:
  - Erste Alternative wird bevorzugt

```
infixl 3 ‘alt‘
alt :: Parse α β→ Parse α β→ Parse α β
alt p1 p2 i = p1 i ++ p2 i
```

- Sequenzierung:
  - Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
(>*>) :: Parse α β→ Parse α γ→ Parse α (β, γ)
(>*>) p1 p2 i =
  concatMap (λ(b, r)→
     map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

Basiskombinatoren: use

- map für Parser (Rückgabe weiterverarbeiten):

```
infix 4 `use`, `use2`
use :: Parse α β→ (β→ γ)→ Parse α γ
use p f i = map (λ(o, r)→ (f o, r)) (p i)

use2 :: Parse α (β, γ)→ (β→ γ→ δ)→ Parse α δ
use2 p f = use p (uncurry f)
```

- Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse α β→ Parse α γ→ Parse α γ
(>*) :: Parse α β→ Parse α γ→ Parse α β
p1 *> p2 = p1 >*> p2 `use` snd
p1 >* p2 = p1 >*> p2 `use` fst
```

Abgeleitete Kombinatoren

- Listen: $A^* ::= AA^* \mid \varepsilon$

```
list :: Parse α β→ Parse α [β]
list p = p >*> list p ‘use2‘ (:)
         ‘alt‘ suceed []
```

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

```
some :: Parse α β→ Parse α [β]
some p = p >*> list p ‘use2‘ (:)
```

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

Verkapselung

- Hauptfunktion:
  - Eingabe muß vollständig parsiert werden
  - Auf Mehrdeutigkeit prüfen

```
parse :: Parse α β→ [α]→ Either String β
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

$$
\begin{array}{rcl}
\textit{Expr} & ::= & \textit{Term}\ \texttt{+}\ \textit{Term} \mid \textit{Term} \\
\textit{Term} & ::= & \textit{Factor} \ast \textit{Factor} \mid \textit{Factor} \\
\textit{Factor} & ::= & \textit{Variable} \mid \texttt{(}\textit{Expr}\texttt{)} \\
\textit{Variable} & ::= & \textit{Char}^{+} \\
\textit{Char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term} + \mathit{Expr} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor} * \mathit{Term} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Variable} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \mathtt{a} \mid \cdots \mid \mathtt{z} \mid \mathtt{A} \mid \cdots \mid \mathtt{Z}
\end{array}
$$

- 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:

$$\begin{aligned} \mathit{Factor} &::= \mathit{Variable} \mid \mathit{Number} \mid \ldots \\ \mathit{Number} &::= \mathit{Digit}^{+} \\ \mathit{Digit} &::= \mathsf{0} \mid \cdots \mid \mathsf{9} \end{aligned}$$

- Eine einfache Eingabesprache:

$$\mathit{Input} ::= \texttt{!}\ \mathit{Variable}\ \texttt{=}\ \mathit{Expr} \mid \texttt{\$}\ \mathit{Expr}$$

- Eine Auswertungsfunktion:

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

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

```haskell
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 (λ(b, r)→
     map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

→ Typinferenz!