Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 26.01.2011: Kombinatoren

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Effizient Funktional Programmieren
  - Fallstudie: Kombinatoren
  - Eine Einführung in Scala
  - Rückblich & Ausblick

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)

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

  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)

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

  S, K, I sind Kombinatoren

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

Kombinatoren als Entwurfsmuster

- 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

Beispiele

- Parserkombinatoren
- Grafikkombinatoren mit der HGL
- Grafikkombinatoren mit tinySVG

Beispiel #1: 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?

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

- Parametrisiert über Eingabetyp (Token) a und Ergebnis b

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse a b} = [\text{a}] \rightarrow \text{b}$$

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

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse a b} = [\text{a}] \rightarrow (\text{b}, [\text{a}])$$

- 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?

$$\textbf{type}\ \text{Parse a b} = [\text{a}] \rightarrow [(\text{b}, [\text{a}])]$$

- 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?

$$\textbf{type}\ \text{Parse a b} = [\text{a}] \rightarrow [(\text{b}, [\text{a}])]$$

- Parametrisiert über Eingabetyp (Token) a und Ergebnis b
- Parser übersetzt Token in abstrakte Syntax
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren
- Beispiel: "a+b*c" $\rightsquigarrow$ [
  (Var "a", "+b*c"),
  (Plus (Var "a") (Var "b") , "*c"),
  (Plus (Var "a") (Times (Var "b") (Var "c")), "")]

Basisparser

- Erkennt nichts:

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

- Erkennt alles:

```
suceed :: b→ Parse a b
suceed 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, suceed 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 (λ(b, r) →
    map (λ(c, s) → ((b, c), s)) (p2 r)) (p1 i)
```

Basiskombinatoren: use

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

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

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

```
some :: Parse a b → Parse a [b]
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 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

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

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.

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.

Beispiel #2:Die Haskell Graphics Library HGL

- Kompakte Grafikbücherei für einfache Grafiken und Animationen.
- Gleiche Schnittstelle zu X Windows (X11) und Microsoft Windows.
- Bietet:
  - Fenster
  - verschiedene Zeichenfunktionen
  - Unterstützung für Animation

Übersicht HGL

- Grafiken

```
type Graphic
```

- Atomare Grafiken:
  - Ellipsen, Linien, Polygone, ...
- Modifikation mit Attributen:
  - Pinsel, Stifte und Textfarben
  - Farben
- Kombination von Grafiken
  - Überlagerung

Basisdatentypen

- Winkel (Grad, nicht Bogenmaß)

```
type Angle     = Double
```

- Dimensionen (Pixel)

```
type Dimension = Int
```

- Punkte (Ursprung: links oben)

```
type Point     = (Dimension, Dimension)
```

Atomare Grafiken (1)

- Ellipse (gefüllt) innerhalb des gegeben Rechtecks

```
ellipse :: Point → Point → Graphic
```

- Ellipse (gefüllt) innerhalb des Parallelograms:

```
shearEllipse :: Point→ Point→ Point→ Graphic
```

- Bogenabschnitt einer Ellipse (math. positiven Drehsinn):

```
arc :: Point→ Point→ Angle→ Angle→ Graphic
```

Atomare Grafiken (2)

- Strecke, Streckenzug:

```
line     :: Point → Point → Graphic
polyline :: [Point] → Graphic
```

- Polygon (gefüllt)

```
polygon :: [Point] → Graphic
```

- Text:

```
text :: Point → String → Graphic
```

- Leere Grafik:

```
emptyGraphic :: Graphic
```

Modifikation von Grafiken

- Andere Fonts, Farben, Hintergrundfarben, . . . :

```
withFont          :: Font      → Graphic → Graphic
withTextColor     :: RGB       → Graphic → Graphic
withBkColor       :: RGB       → Graphic → Graphic
withBkMode        :: BkMode    → Graphic → Graphic
withPen           :: Pen       → Graphic → Graphic
withBrush         :: Brush     → Graphic → Graphic
withRGB           :: RGB       → Graphic → Graphic
withTextAlignment :: Alignment → Graphic → Graphic
```

Farben

- Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

- Benannte Farben sind einfach colorTable :: Array Color RGB

- Dazu Modifikator:

```
withColor :: Color → Graphic → Graphic
withColor c = withRGB (colorTable ! c)
```

Kombination von Grafiken

- Überlagerung (erste über zweiter):

```
overGraphic :: Graphic → Graphic → Graphic
```

- Verallgemeinerung:

```
overGraphics :: [Graphic] → Graphic
overGraphics = foldr overGraphic emptyGraphic
```

Fenster

- Elementare Funktionen:

```
getGraphic :: Window → IO Graphic
setGraphic :: Window → Graphic → IO ()
```

- Abgeleitetete Funktionen:
  - In Fenster zeichnen:

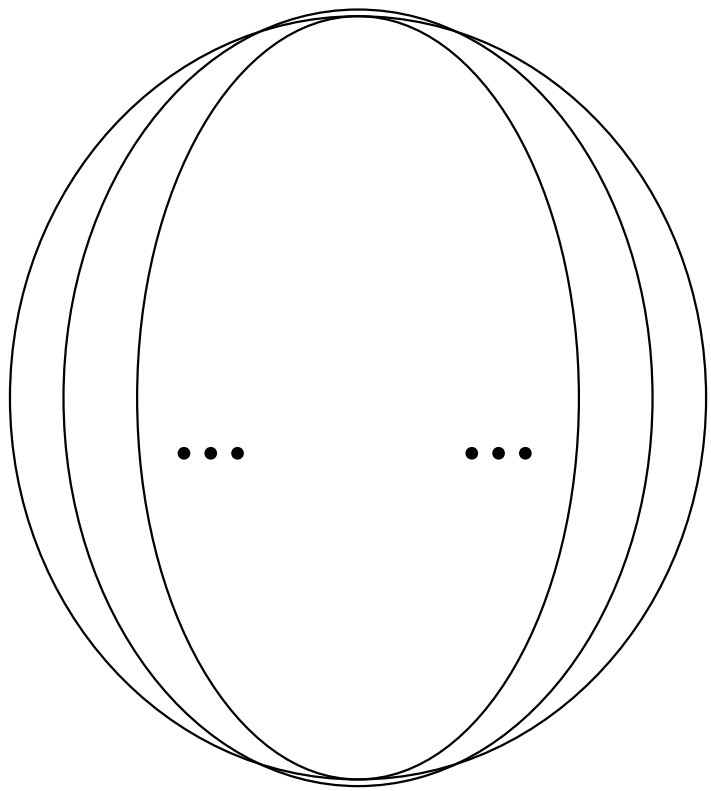
```
drawInWindow :: Window → Graphic → IO ()
drawInWindow w g = do
  old ← getGraphic w
  setGraphic w (g `overGraphic` old)
```

  - Grafik löschen

```
clearWindow :: Window → IO ()
clearWindow w = setGraphic w emptyGraphic
```

Ein einfaches Beispiel: der Ball

- Ziel: einen gestreiften Ball zeichen
- Algorithmus: als Folge von konzentrischen Ellipsen
  - Start mit Eckpunkten (x_1, y_1) und (x_2, y_2).
  - Verringerung von x um Δ_x, y bleibt gleich.
  - Dabei Farbe verändern.



Ein einfaches Beispiel: Der Ball

- Liste aller Farben cols
- Listen der x-Position (y-Position ist konstant), $\Delta_x = 25$
- Hilfsfunktion drawEllipse

```
drawBall :: Point→ Point→ Graphic
drawBall (x1, y1) (x2, y2) =
  let cols = cycle [Red, Green, Blue]
      midx = (x2− x1) `div` 2
      xls  = [x1, x1+ 25 .. midx]
      xrs  = [x2, x2− 25 .. midx]
      drawEllipse c xl xr = withColor c $
                            ellipse (xl, y1) (xr, y2)
      gs   = zipWith3 drawEllipse cols xls xrs
  in  overGraphics (reverse gs)
```

Ein einfaches Beispiel: Der Ball

- Hauptprogramm (Zeigen)

```
main :: IO ()
main = runGraphics $ do
    w ← openWindow "Balls!" (500,500)
    drawInWindow w $ drawBall (25, 25) (485, 485)
    getKey w
    closeWindow w
```

Animation

Alles dreht sich, alles bewegt sich...

- Animation: über der Zeit veränderliche Grafik
- Unterstützung von Animationen in HGL:
  - `Timer` ermöglichen getaktete Darstellung
  - Gepufferte Darstellung ermöglicht flickerfreie Darstellung
- Öffnen eines Fensters mit Animationsunterstützung:
  - Initiale Position, Grafikzwischenpuffer, Timer-Takt in Millisekunden

```
openWindowEx :: Title → Maybe Point → Size →
                RedrawMode → Maybe Time → IO Window
data RedrawMode
  = Unbuffered | DoubleBuffered
```

Der springende Ball

- Ball hat Position und Geschwindigkeit:

```
data Ball = Ball { p :: Point,
                   v :: Point }
```

- Ball zeichnen: Roter Kreis an Position $\vec{p}$

```
drawBall :: Ball→ Graphic
drawBall (Ball {p= p}) =
  withColor Red (circle p 15)
```

- Kreis zeichnen:

```
circle :: Point→ Int→ Graphic
circle (px, py) r = ellipse (px− r, py− r) (px+ r, p
```

Bewegung des Balles

- Geschwindigkeit $\vec{v}$ zu Position $\vec{p}$ addieren
- In X-Richtung: modulo Fenstergröße 500
- In Y-Richtung: wenn Fensterrand 500 erreicht, Geschwindigkeit invertieren
- Geschwindigkeit in Y-Richtung nimmt immer um 1 ab

```
move (Ball {p= (px, py), v= (vx, vy)})=
  Ball {p= (px', py'), v= (vx, vy')} where
    px' = (px+ vx) `mod` 500
    py0 = py+ vy
    py' = if py0> 500 then 500-(py0-500) else py0
    vy' = (if py0> 500 then -vy else vy)+ 1
```

Der springende Ball

- Hauptschleife: Ball zeichnen, auf Tick warten, Folgeposition

```
loop w b =
  do setGraphic w (drawBall b)
     getWindowTick w
     loop w (move b)
```

- Hauptprogram: Fenster öffnen, Starten der Hauptschleife

```
main = runGraphics $
  do w← openWindowEx "Bounce!" Nothing (500, 500)
                     DoubleBuffered (Just 30)
     loop w (Ball{p=(0, 10), v= (5, 0)})
```

Zusammenfassung HGL

- Abstrakte und portable Grafikprogrammierung
- Verkapselung von systemnaher Schnittstelle durch Kombinatoren
- Kombinatoransatz: Kombination elementarer Grafiken zu komplexen Grafikprogrammen
- Rudimentäre Unterstützung von Animation durch Timer und Puffer
- Kombinatoransatz hier:

```
type Time = Int
type Animation = Int → Graphic
```

Beispiel #3: tinySVG

- Scalable Vector Graphics (SVG)
  - XML-Standard für Vektorgrafiken
  - Unterstützt Vektorgrafiken (Pfade), Rastergrafiken und Text
  - Ein Kreis: `<circle x="20" y="30" r="50" />`
- Übersetzung in Kombinatorbücherei in Haskell (tinySVG):
  - Elementare Operationen:

```
circle :: Point → Double → Graphics
line   :: Point → Point → Graphics
```

  - Kombinatoren zum Transformieren:

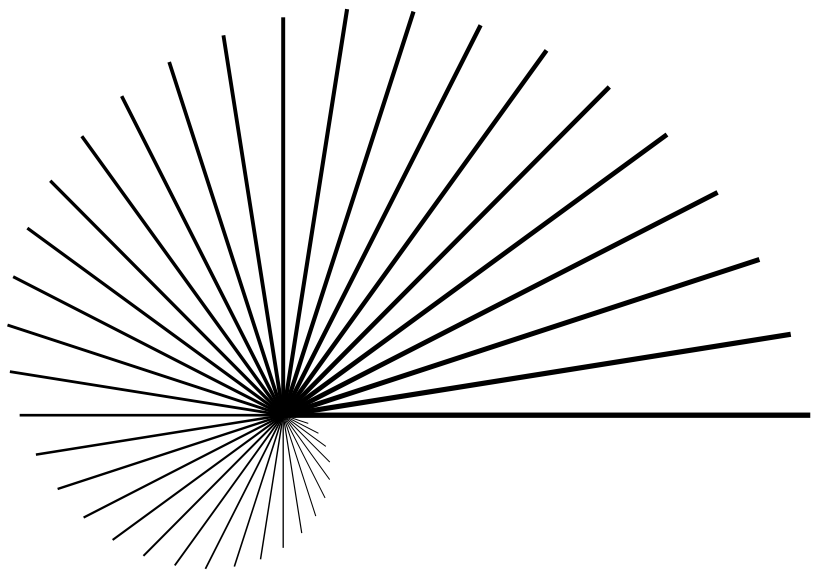
```
group  :: [Graphics] → Graphics
rotate :: Double → Graphics → Graphics
scale  :: Double → Graphics → Graphics
```

  - Ausgabe:

```
toXML :: Double → Double → Graphics → String
```

tinySVG in Aktion

```
snailShell :: Int → Graphics
snailShell n =
  let rs = [fromInt i / fromInt n | i ← [1..n]]
      theLine = line (pt' 0 0) (pt' 100 0)
      lines = [rotate (r*360) $ scale r theLine
              | r ← rs]
  in group lines
```



Zusammenfassung

- Kombinatoransatz:
  - Einfache Basisoperationen
  - Wenige Kombinationsoperatoren
  - Ideal in funktionalen Sprachen, generell nützlich
- Parserkombinatoren:
  - Von Grund auf in Haskell
  - Kombinatoren abstrahieren über Implementation
- Grafik mit HGL:
  - Verkapselung von Systemschnittstellen
  - Kombinatoren abstrahieren Systemschnittstellen
- Grafik mit tinySVG:
  - Kombinatoren abstrahieren über XML-Syntax