

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 2 vom 23.10.2012: Funktionen und Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Rev. 1843

1 [35]

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

2 [35]

Inhalt

- ▶ Auswertungsstrategien
 - ▶ Striktheit
- ▶ Definition von Funktionen
 - ▶ Syntaktische Feinheiten
- ▶ Definition von Datentypen
 - ▶ Aufzählungen
 - ▶ Produkte

3 [35]

Auswertungsstrategien

```
inc :: Int -> Int      double :: Int -> Int
inc x = x + 1          double x = 2*x
```

- ▶ Reduktion von `inc (double (inc 3))`
- ▶ Von außen nach innen (outermost-first):
 - `inc (double (inc 3))` \rightsquigarrow `double (inc 3) + 1`
 - \rightsquigarrow `2*(inc 3) + 1`
 - \rightsquigarrow `2*(3 + 1) + 1`
 - \rightsquigarrow `2*4 + 1` \rightsquigarrow 9
- ▶ Von innen nach außen (innermost-first):
 - `inc (double (inc 3))` \rightsquigarrow `inc (double (3+1))`
 - \rightsquigarrow `inc (2*(3 + 1))`
 - \rightsquigarrow `(2*(3 + 1)) + 1`
 - \rightsquigarrow `2*4 + 1` \rightsquigarrow 9

4 [35]

Konfluenz und Termination

Sei \rightsquigarrow^* die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

\rightsquigarrow^* ist **konfluent** gdw:
Für alle r, s, t mit $r \rightsquigarrow^* s, r \rightsquigarrow^* t$ gibt es u so dass $s \rightsquigarrow^* u, t \rightsquigarrow^* u$.

Definition (Termination)

\rightsquigarrow ist **terminierend** gdw. es keine unendlichen Ketten gibt:
 $t_1 \rightsquigarrow t_2 \rightsquigarrow t_3 \rightsquigarrow \dots t_n \rightsquigarrow \dots$

5 [35]

Auswertungsstrategien

Theorem (Konfluenz)

Funktionale Programme sind für jede Auswertungsstrategie **konfluent**.

Theorem (Normalform)

Terminierende funktionale Programme werten unter jeder Auswertungsstrategie jeden Ausdruck zum gleichen Wert aus (der **Normalform**).

- ▶ Auswertungsstrategie für **nicht-terminierende** Programme relevant
- ▶ Nicht-Termination **nötig** (Turing-Mächtigkeit)

6 [35]

Auswirkung der Auswertungsstrategie

- ▶ Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
- ▶ Innermost-first entspricht **call-by-value**, strikte Auswertung
- ▶ Beispiel:

```
repeat :: Int -> String -> String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`

7 [35]

Striktheit

Definition (Striktheit)

Funktion f ist **strikt** \iff Ergebnis ist undefiniert
sobald ein Argument undefiniert ist

- ▶ **Denotationelle** Eigenschaft (nicht operational)
- ▶ Java, C etc. sind **call-by-value** (nach Sprachdefinition) und damit **strikt**
- ▶ Haskell ist **nicht-strikt** (nach Sprachdefinition)
 - ▶ Meisten Implementierungen nutzen **verzögerte Auswertung**
- ▶ Fallunterscheidung ist **immer** nicht-strikt

8 [35]

Haskell-Syntax: Funktionsdefinitionsdefinition

Generelle Form:

► **Signatur:**

```
max :: Int → Int → Int
```

► **Definition**

```
max x y = if x < y then y else x
```

- Kopf, mit Parametern
- Rumpf (evtl. länger, mehrere Zeilen)
- Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf
- Was gehört zum Rumpf (Geltungsbereich)?

9 [35]

Haskell-Syntax: Charakteristika

► **Leichtgewichtig**

- Wichtigstes Zeichen:
- Funktionsapplikation: `f a`
- Keine Klammern
- Abseitsregel: Gültigkeitsbereich durch Einrückung
- Keine Klammern
- Auch in anderen Sprachen (Python, Ruby)

10 [35]

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

```
f x1 x2 ... xn = E
```

- **Geltungsbereich** der Definition von `f`:
alles, was gegenüber `f` eingerückt ist.

► **Beispiel:**

```
f x = hier faengts an  
    und hier gehts weiter  
    immer weiter  
g y z = und hier faengt was neues an
```

- Gilt auch verschachtelt.
- Kommentare sind **passiv**

11 [35]

Haskell-Syntax II: Kommentare

- Pro Zeile: Ab `--` bis Ende der Zeile

```
f x y = irgendwas -- und hier der Kommentar!
```

- Über mehrere Zeilen: Anfang `{-`, Ende `-}`

```
{-  
  Hier fängt der Kommentar an  
  erstreckt sich über mehrere Zeilen  
  bis hier -}  
f x y = irgendwas
```

- Kann geschachtelt werden.

12 [35]

Haskell-Syntax III: Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen ...

```
f x y = if B1 then P else  
        if B2 then Q else ...
```

... **bedingte Gleichungen:**

```
f x y  
| B1 = ...  
| B2 = ...
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: **Laufzeitfehler!** Deshalb:

```
| otherwise = ...
```

13 [35]

Haskell-Syntax IV: Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y  
| g = P y  
| otherwise =  
Q where  
  y = M  
  f x = N x  
f x y =  
  let y = M  
      f x = N x  
  in if g then P y  
      else Q
```

- `f, y, ...` werden **gleichzeitig** definiert (Rekursion!)
- Namen `f, y` und Parameter `(x)` **überlagern** andere
- Es gilt die **Abseitsregel**
 - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

14 [35]

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand ↔ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

- `Int` - ganze Zahlen als Maschinenworte (≥ 31 Bit)
- `Integer` - beliebig große ganze Zahlen
- `Rational` - beliebig genaue rationale Zahlen
- `Float, Double` - Fließkommazahlen (reelle Zahlen)

15 [35]

Ganze Zahlen: `Int` und `Integer`

- Nützliche Funktionen (**überladen**, auch für `Integer`):

```
+, *, ^, - :: Int → Int → Int  
abs :: Int → Int -- Betrag  
div, quot :: Int → Int → Int  
mod, rem :: Int → Int → Int
```

Es gilt $(\text{div } x \ y) * y + \text{mod } x \ y == x$

- Vergleich durch `==, /=, <=, <, ...`
- **Achtung:** Unäres Minus
 - Unterschied zum Infix-Operator -
 - Im Zweifelsfall klammern: `abs (-34)`

16 [35]

Fließkommazahlen: Double

- ▶ Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
 - ▶ Logarithmen, Wurzel, Exponentiation, π und e , trigonometrische Funktionen
- ▶ Konversion in ganze Zahlen:
 - ▶ fromIntegral :: Int, Integer -> Double
 - ▶ fromInteger :: Integer -> Double
 - ▶ round, truncate :: Double -> Int, Integer
 - ▶ Überladungen mit Typnotation auflösen:


```
round (fromInt 10) :: Int
```
- ▶ Rundungsfehler!

17 [35]

Alphanumerische Basisdatentypen: Char

- ▶ Notation für einzelne Zeichen: 'a', ...
- ▶ Nützliche Funktionen:


```
ord :: Char -> Int
chr  :: Int  -> Char

toLower :: Char -> Char
toUpper :: Char -> Char
isDigit  :: Char -> Bool
isAlpha  :: Char -> Bool
```
- ▶ Zeichenketten: String

18 [35]

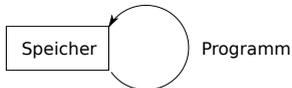
Datentypen als Modellierungskonstrukt

Programme **manipulieren** ein **Modell** (der Umwelt)

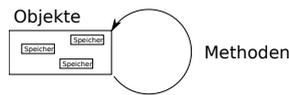
- ▶ Funktionale Sicht:



- ▶ Imperative Sicht:



- ▶ Objektorientierte Sicht:



19 [35]

Beispiel: Uncle Bob's Auld-Time Grocery Shoppe

| | | | |
|-----------|--------------|-------|---------|
| Äpfel | Boskoop | 55 | ct/Stk |
| | Cox Orange | 60 | ct/Stk |
| | Granny Smith | 50 | ct/Stk |
| Eier | | 20 | ct/Stk |
| Käse | Gouda | 14,50 | €/kg |
| | Appenzeller | 22,70 | €/kg |
| Schinken | | 1,99 | €/100 g |
| Mettwurst | | 1,59 | €/100 g |
| Milch | | 0,69 | €/l |
| | Bio | 1,19 | €/l |

20 [35]

Typkonstruktoren

- ▶ Aufzählungen
- ▶ Produkt
- ▶ Rekursion
- ▶ Funktionsraum

21 [35]

Aufzählungen

- ▶ Aufzählungen: Menge von **disjunkten** Konstanten

$$\text{Apfel} = \{\text{Boskoop}, \text{Cox}, \text{Smith}\}$$

$$\text{Boskoop} \neq \text{Cox}, \text{Cox} \neq \text{Smith}, \text{Boskoop} \neq \text{Smith}$$

- ▶ Genau drei unterschiedliche Konstanten
- ▶ Funktion mit Wertebereich *Apfel* muss drei Fälle unterscheiden
- ▶ Beispiel: $\text{preis} : \text{Apfel} \rightarrow \mathbb{N}$ mit

$$\text{preis}(a) = \begin{cases} 55 & a = \text{Boskoop} \\ 60 & a = \text{Cox} \\ 50 & a = \text{Smith} \end{cases}$$

22 [35]

Aufzählung und Fallunterscheidung in Haskell

- ▶ Definition

```
data Apfel = Boskoop | CoxOrange | GrannySmith
```

- ▶ Implizite Deklaration der **Konstruktoren** $\text{Boskoop} :: \text{Apfel}$ als Konstanten
- ▶ Großschreibung der Konstruktoren

- ▶ Fallunterscheidung:

```
apreis :: Apfel -> Int
apreis a = case a of
  Boskoop -> 55
  CoxOrange -> 60
  GrannySmith -> 50
```

```
data Farbe = Rot | Grn
farbe :: Apfel -> Farbe
farbe d =
  case d of
    GrannySmith -> Grn
    _ -> Rot
```

23 [35]

Fallunterscheidung in der Funktionsdefinition

- ▶ Abkürzende Schreibweisen (**syntaktischer Zucker**):

$$\begin{array}{l} f\ c_1 == e_1 \\ \dots \\ f\ c_n == e_n \end{array} \quad \rightarrow \quad f\ x == \text{case } x \text{ of } \begin{array}{l} c_1 \rightarrow e_1, \\ \dots \\ c_n \rightarrow e_n \end{array}$$

- ▶ Damit:

```
preis :: Apfel -> Int
preis Boskoop = 55
preis CoxOrange = 60
preis GrannySmith = 50
```

24 [35]

Der einfachste Aufzählungstyp

- ▶ **Einfachste** Aufzählung: Wahrheitswerte

$Bool = \{True, False\}$

- ▶ Genau zwei unterschiedliche Werte
- ▶ **Definition** von Funktionen:

- ▶ Wertetabellen sind explizite Fallunterscheidungen

| | | | |
|----------|--------------|--------------|---|
| \wedge | <i>true</i> | <i>false</i> | <i>true</i> \wedge <i>true</i> = <i>true</i> |
| | <i>true</i> | <i>false</i> | <i>true</i> \wedge <i>false</i> = <i>false</i> |
| | <i>false</i> | <i>true</i> | <i>false</i> \wedge <i>true</i> = <i>false</i> |
| | <i>false</i> | <i>false</i> | <i>false</i> \wedge <i>false</i> = <i>false</i> |

25 [35]

Wahrheitswerte: Bool

- ▶ **Vordefiniert** als

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

- ▶ Vordefinierte **Funktionen**:

```
not :: Bool → Bool      — Negation
&&  :: Bool → Bool → Bool — Konjunktion
||  :: Bool → Bool → Bool — Disjunktion
```

- ▶ **Konjunktion** definiert als

```
a && b = case a of True  → b
                False → False
```

- ▶ $\&\&$, $\|\|$ sind rechts nicht strikt
 - ▶ $1 == 0 \&\& \text{div } 10 == 0 \rightsquigarrow \text{False}$
- ▶ if then else als syntaktischer Zucker:

```
if b then p else q → case b of True  → p
                        False → q
```

26 [35]

Beispiel: Ausschließende Disjunktion

- ▶ Mathematische Definition:

```
exOr :: Bool → Bool → Bool
exOr x y = (x || y) && (not (x && y))
```

- ▶ Alternative 1: explizite Wertetabelle:

```
exOr False False = False
exOr True  False = True
exOr False True  = True
exOr True  True  = False
```

- ▶ Alternative 2: Fallunterscheidung auf ersten Argument

```
exOr True  y = not y
exOr False y = y
```

- ▶ Was ist am **besten**?
 - ▶ Effizienz, Lesbarkeit, Striktheit

27 [35]

Produkte

- ▶ Konstruktoren können **Argumente** haben
- ▶ Beispiel: Ein **Datum** besteht aus **Tag**, **Monat**, **Jahr**
- ▶ Mathematisch: Produkt (Tupel)

$Date = \{Date(n, m, y) \mid n \in \mathbb{N}, m \in Month, y \in \mathbb{N}\}$
 $Month = \{Jan, Feb, Mar, \dots\}$

- ▶ **Funktionsdefinition**:

- ▶ Konstruktorenargumente sind **gebundene Variablen**

```
year(D(n, m, y)) = y
day(D(n, m, y))  = n
```

- ▶ Bei der **Auswertung** wird **gebundene Variable** durch konkretes **Argument** ersetzt

28 [35]

Produkte in Haskell

- ▶ Konstruktoren mit **Argumenten**

```
data Date = Date Int Month Int
data Month = Jan | Feb | Mar | Apr | May | Jun
            | Jul | Aug | Sep | Oct | Nov | Dec
```

- ▶ **Beispielwerte**:

```
today      = Date 23 Oct 2012
bloomsday  = Date 16 Jun 1904
```

- ▶ Über **Fallunterscheidung** Zugriff auf **Argumente** der Konstruktoren:

```
day :: Date → Int
year :: Date → Int
day d = case d of Date t m y → t
year (Date d m y) = y
```

29 [35]

Beispiel: Tag im Jahr

- ▶ **Tag im Jahr**: Tag im laufenden Monat plus Summe der Anzahl der Tage der vorherigen Monate

```
yearDay :: Date → Int
yearDay (Date d m y) = d + sumPrevMonths m where
  sumPrevMonths :: Month → Int
  sumPrevMonths Jan = 0
  sumPrevMonths m   = daysInMonth (prev m) y +
    sumPrevMonths (prev m)
```

- ▶ **Tage im Monat** benötigt **Jahr** als Argument (Schaltjahr!)

```
daysInMonth :: Month → Int → Int
prev :: Month → Month
```

- ▶ **Schaltjahr**: Gregorianischer Kalender

```
leapyear :: Int → Bool
leapyear y = if mod y 100 == 0 then mod y 400 == 0
            else mod y 4 == 0
```

30 [35]

Beispiel: Produkte in Bob's Shoppe

- ▶ Käsesorten und deren Preise:

```
data Kaese = Gouda | Appenzeller
```

```
kpreis :: Kaese → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

- ▶ Alle Artikel:

```
data Artikel =
  Apfel Apfel | Eier
  | Kaese Kaese | Schinken
  | Salami      | Milch Bool
```

31 [35]

Beispiel: Produkte in Bob's Shoppe

- ▶ **Mengenangaben**:

```
data Menge = Stueck Int | Gramm Int
            | Kilo Double | Liter Double
```

- ▶ **Der Preis** und seine Berechnung:

```
data Preis = Cent Int | Ungueltig
```

```
preis :: Artikel → Menge → Preis
preis (Apfel a) (Stueck n) = Cent (n * apreis a)
preis Eier (Stueck n)      = Cent (n * 20)
preis (Kaese k) (Kilo kg)  = Cent (round (kg *
                                         kpreis k))
preis Schinken (Gramm g)  = Cent (g * 199)
preis Salami (Gramm g)    = Cent (g * 159)
preis (Milch bio) (Liter l) =
  Cent (round (l * if not bio then 69 else 119))
```

32 [35]

Auswertung der Fallunterscheidung

- ▶ Argument der Fallunterscheidung wird **nur soweit nötig** ausgewertet

- ▶ Beispiel:

```
data Foo = Foo Int | Bar

f :: Foo -> Int
f foo = case foo of Foo i -> i; Bar -> 0

g :: Foo -> Int
g foo = case foo of Foo i -> 9; Bar -> 0

add :: Foo -> Foo -> Foo
add (Foo i) (Foo j) = Foo (i + j)
add _ _ = Bar
```

33 [35]

Der Allgemeine Fall: Algebraische Datentypen

Definition eines **algebraischen Datentypen** T:

data T = C₁ t_{1,1} ... t_{1,k₁}

...

| C_n t_{n,1} ... t_{n,k_n}

- ▶ Konstruktoren C₁, ..., C_n sind **disjunkt**:
 $C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$

- ▶ Konstruktoren sind **injektiv**:
 $C x_1 \dots x_n = C y_1 \dots y_n \implies x_i = y_i$

- ▶ Konstruktoren **erzeugen** den Datentyp:
 $\forall x \in T. x = C_i y_1 \dots y_m$

Diese Eigenschaften machen **Fallunterscheidung** möglich.

Rekursion? \rightsquigarrow **Nächste Vorlesung!**

34 [35]

Zusammenfassung

- ▶ **Striktheit**
 - ▶ Haskell ist **spezifiziert** als nicht-strikt
- ▶ Numerische Basisdatentypen:
 - ▶ Int, Integer, Rational und Double
- ▶ Alphanumerische Basisdatentypen: Char
- ▶ Datentypen und Funktionsdefinition dual
 - ▶ **Aufzählungen** — **Fallunterscheidung**
 - ▶ **Produkte** — **Projektion**
- ▶ **Algebraische Datentypen**
 - ▶ **Drei wesentliche Eigenschaften** der Konstruktoren
- ▶ Wahrheitswerte Bool
- ▶ **Nächste Vorlesung**: Rekursive Datentypen

35 [35]