

Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 03.11.2010: Funktionen und Datentypen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Rev. 1167

1 [36]

Inhalt

- ▶ Auswertungsstrategien
 - ▶ Striktheit
- ▶ Definition von **Funktionen**
 - ▶ Syntaktische Feinheiten
- ▶ Definition von **Datentypen**
 - ▶ Aufzählungen
 - ▶ Produkte
- ▶ **Basisdatentypen:**
 - ▶ Wahrheitswerte, numerische Typen, alphanumerische Typen

2 [36]

Fahrplan

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

3 [36]

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 [36]

Auswertungsstrategien und Konfluenz

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)

5 [36]

Auswirkung der Auswertungsstrategie

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

```
div :: Int -> Int -> Int
Ganzzahlige Division, undefiniert für div n 0

mult :: Int -> Int -> Int
mult n m = if n == 0 then 0
           else (mult (n - 1) m) + m
```
- ▶ Auswertung von `mult 0 (div 1 0)`

6 [36]

Striktheit

Definition (Striktheit)

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

- ▶ **Semantische** Eigenschaft (nicht operational)
- ▶ Standard ML, Java, C etc. sind **strikt** (nach Sprachdefinition)
- ▶ Haskell ist **nicht-strikt** (nach Sprachdefinition)
 - ▶ Meisten **Implementationen** nutzen **verzögerte Auswertung**
- ▶ Fallunterscheidung ist **immer** nicht-strikt

7 [36]

Wie definiere ich eine Funktion?

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)?

8 [36]

Haskell-Syntax: Charakteristika

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

9 [36]

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**

10 [36]

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.

11 [36]

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 = ...
```

12 [36]

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
```

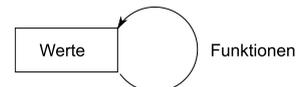
- ▶ `f`, `y`, ... werden **gleichzeitig** definiert (Rekursion!)
- ▶ Namen `f`, `y` und Parameter (`x`) **überlagern** andere
- ▶ Es gilt die **Abseitsregel**
 - ▶ Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

13 [36]

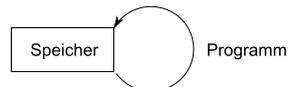
Datentypen als Modellierungskonstrukt

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

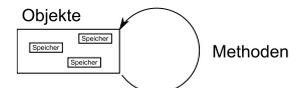
- ▶ **Funktionale Sicht**:



- ▶ **Imperative Sicht**:



- ▶ **Objektorientierte Sicht**:



14 [36]

Datentypen, Funktionen und Beweise

- ▶ Datentypen konstruieren **Werte**
- ▶ Funktionen definieren **Berechnungen**
- ▶ Berechnungen haben **Eigenschaften**
- ▶ Dualität:
Datentypkonstruktor \longleftrightarrow Definitionskonstrukt \longleftrightarrow Beweiskonstrukt

15 [36]

Typkonstruktoren

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

16 [36]

Aufzählungen

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

$$\text{Days} = \{\text{Mon}, \text{Tue}, \text{Wed}, \text{Thu}, \text{Fri}, \text{Sat}, \text{Sun}\}$$

$$\text{Mon} \neq \text{Tue}, \text{Mon} \neq \text{Wed}, \text{Tue} \neq \text{Thu}, \text{Wed} \neq \text{Sun} \dots$$

- ▶ Genau sieben unterschiedliche Konstanten
- ▶ Funktion mit Wertebereich *Days* muss sieben Fälle unterscheiden
- ▶ Beispiel: $\text{weekend} : \text{Days} \rightarrow \text{Bool}$ mit

$$\text{weekend}(d) = \begin{cases} \text{true} & d = \text{Sat} \vee d = \text{Sun} \\ \text{false} & d = \text{Mon} \vee d = \text{Tue} \vee d = \text{Wed} \vee \\ & d = \text{Thu} \vee d = \text{Fri} \end{cases}$$

17 [36]

Aufzählung und Fallunterscheidung in Haskell

- ▶ **Definition**

```
data Days = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

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

- ▶ **Fallunterscheidung:**

```
weekend :: Days -> Bool
weekend d = case d of
  Sat -> True
  Sun -> True
  Mon -> False
  Tue -> False
  Wed -> False
  Thu -> False
  Fri -> False
```

```
weekend d =
case d of
  Sat -> True
  Sun -> True
  _ -> False
```

18 [36]

Fallunterscheidung in der Funktionsdefinition

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

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

- ▶ Damit:

```
weekend :: Days -> Bool
weekend Sat = True
weekend Sun = True
weekend _ = False
```

19 [36]

Der einfachste Aufzählungstyp

- ▶ **Einfachste** Aufzählung: Wahrheitswerte

$$\text{Bool} = \{\text{True}, \text{False}\}$$

- ▶ Genau zwei unterschiedliche Werte

- ▶ **Definition** von Funktionen:

- ▶ Wertetabellen sind explizite Fallunterscheidungen

\wedge	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

```

true ^ true = true
true ^ false = false
false ^ true = false
false ^ false = false

```

20 [36]

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
 - ▶ $1 == 0 \ \&\& \ \text{div } 1 \ 0 == 0 \rightsquigarrow \text{False}$
- ▶ **if then else** als syntaktischer Zucker:

$$\text{if } b \text{ then } p \text{ else } q \longrightarrow \text{case } b \text{ of } \begin{array}{l} \text{True} \rightarrow p \\ \text{False} \rightarrow q \end{array}$$

21 [36]

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

22 [36]

Produkte

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

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

- ▶ **Funktionsdefinition:**

- ▶ Konstruktargumente sind gebundene Variablen

$$\begin{array}{l} \text{year}(D(n, m, y)) = y \\ \text{day}(D(n, m, y)) = n \end{array}$$

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

23 [36]

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 5 Nov 2008
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
```

24 [36]

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
```

25 [36]

Fallunterscheidung und Produkte

- ▶ Beispiel: geometrische Objekte

- ▶ Dreieck, gegeben durch Kantenlänge

- ▶ Kreis, gegeben durch Radius

- ▶ Rechteck, gegeben durch zwei Kantenlängen

$$O = \{Tri(a) \mid a \in \mathbb{R}\} \cup \{Circle(r) \mid r \in \mathbb{R}\} \cup \{Rect(a, b) \mid a, b \in \mathbb{R}\}$$

```
data Obj = Tri Double | Circle Double | Rect Double Double
```

- ▶ Berechnung des Umfangs:

```
circ :: Obj -> Double
circ (Tri a) = 3 * a
circ (Circle r) = 2 * pi * r
circ (Rect a b) = 2 * (a + b)
```

26 [36]

Der Allgemeine Fall: Algebraische Datentypen

Definition eines **algebraischen Datentypen** T:

```
data T = C1 t1,1 ... t1,k1
      | ...
      | Cn tn,1 ... tn,kn
```

- ▶ Konstruktoren C_1, \dots, 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.

27 [36]

Beweis von Eigenschaften

- ▶ Eigenschaften von Programmen: **Prädikate**

- ▶ Haskell-Ausdrücke vom Typ Bool

- ▶ Allquantifizierte Aussagen:

wenn $P(x)$ Prädikat, dann ist $\forall x. P(x)$ auch ein Prädikat

- ▶ Sonderfall Gleichungen $s == t$

- ▶ Müssen nicht **ausführbar** sein

28 [36]

Wie beweisen?

- ▶ Gleichungsumformung (equational reasoning)

- ▶ **Fallunterscheidungen**

- ▶ Induktion

- ▶ Wichtig: **formale Notation**

29 [36]

Ein ganz einfaches Beispiel

```
addTwice :: Int -> Int -> Int
addTwice x y = 2 * (x + y)
```

Lemma: $\text{addTwice } x (y + z) = \text{addTwice } (x + y) z$

$$\begin{aligned}
 & \text{addTwice } x (y + z) \\
 &= 2 * (x + (y + z)) \quad \text{--- Def. addTwice} \\
 &= 2 * ((x + y) + z) \quad \text{--- Assoziativität von +} \\
 &= \text{addTwice } (x + y) z \quad \text{--- Def. addTwice} \\
 & \square
 \end{aligned}$$

30 [36]

Fallunterscheidung

```
max, min :: Int -> Int -> Int
max x y = if x < y then y else x
min x y = if x < y then x else y
```

Lemma: $\text{max } x y - \text{min } x y = |x - y|$

$$\begin{aligned}
 & \text{max } x y - \text{min } x y \\
 & \bullet \text{ Fall: } x < y \\
 &= y - \text{min } x y \quad \text{--- Def. max} \\
 &= y - x \quad \text{--- Def. min} \\
 &= |x - y| \quad \text{--- Wenn } x < y, \text{ dann } y - x = |x - y| \\
 & \bullet \text{ Fall: } x \geq y \\
 &= x - \text{min } x y \quad \text{--- Def. max} \\
 &= x - y \quad \text{--- Def. min} \\
 &= |x - y| \quad \text{--- Wenn } x \geq y, \text{ dann } x - y = |x - y| \\
 &= |x - y| \\
 & \square
 \end{aligned}$$

31 [36]

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand \longleftrightarrow 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)

32 [36]

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)`

33 [36]

Fließkommazahlen: Double

- ▶ Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
 - ▶ Logarithmen, Wurzel, Exponentation, 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!**

34 [36]

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`

35 [36]

Zusammenfassung

- ▶ **Striktheit**
 - ▶ Haskell ist **spezifiziert** als nicht-strikt
- ▶ **Datentypen und Funktionsdefinition dual**
 - ▶ **Aufzählungen** — **Fallunterscheidung**
 - ▶ **Produkte**
- ▶ **Funktionsdefinition und Beweis dual**
 - ▶ Beweis durch **Gleichungsumformung**
 - ▶ Programmeigenschaften als **Prädikate**
 - ▶ **Fallunterscheidung** als Beweiskonstrukt
- ▶ **Wahrheitswerte Bool**
- ▶ **Numerische Basisdatentypen**:
 - ▶ `Int, Integer, Rational` und `Double`
- ▶ **Alphanumerische Basisdatentypen**: `Char`
- ▶ **Nächste Vorlesung**: Rekursive Datentypen

36 [36]