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

Inhalt

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

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

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) $\rightsquigarrow$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

$$\begin{aligned} \texttt{inc (double (inc 3))} &\rightsquigarrow \texttt{double (inc 3)}+ 1 \\ &\rightsquigarrow \end{aligned}$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

inc (double (inc 3)) ⇝ double (inc 3)+ 1
⇝ 2*(inc 3)+ 1
⇝

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

inc (double (inc 3)) ⇝ double (inc 3)+ 1
⇝ 2*(inc 3)+ 1
⇝ 2*(3+ 1)+ 1
⇝ 2*4+1 ⇝ 9

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) ⇝ double (inc 3)+ 1
  ⇝ 2*(inc 3)+ 1
  ⇝ 2*(3+ 1)+ 1
  ⇝ 2*4+1 ⇝ 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) ⇝

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) ⇝ double (inc 3)+ 1
  ⇝ 2*(inc 3)+ 1
  ⇝ 2*(3+ 1)+ 1
  ⇝ 2*4+1 ⇝ 9
- Von innen nach außen (innermost-first):

  inc (double (inc 3)) ⇝ inc (double (3+1))
  ⇝

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) ⇝ double (inc 3)+ 1
  ⇝ 2*(inc 3)+ 1
  ⇝ 2*(3+ 1)+ 1
  ⇝ 2*4+1 ⇝ 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) ⇝ inc (double (3+1))
  ⇝ inc (2*(3+ 1))
  ⇝

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) ⇝ double (inc 3)+ 1
  ⇝ 2*(inc 3)+ 1
  ⇝ 2*(3+ 1)+ 1
  ⇝ 2*4+1 ⇝ 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) ⇝ inc (double (3+1))
  ⇝ inc (2*(3+ 1))
  ⇝ (2*(3+ 1))+ 1
  ⇝

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) ⇝ double (inc 3)+ 1
  ⇝ 2*(inc 3)+ 1
  ⇝ 2*(3+ 1)+ 1
  ⇝ 2*4+1 ⇝ 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) ⇝ inc (double (3+1))
  ⇝ inc (2*(3+ 1))
  ⇝ (2*(3+ 1))+ 1
  ⇝ 2*4+1 ⇝ 9

Auswertungsstrategien und Konfluenz

Theorem (Konfluenz)

Funktionale Programme sind für jede Auswertungsstrategie konfluent.

Theorem (Normalform)

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

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

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)

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

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 (Geltungsberereich)?

Haskell-Syntax: Charakteristika

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

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

$\mathtt{f\ x_1\ x_2 \ldots x_n = E}$

- 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

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.

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 =...
```

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
```

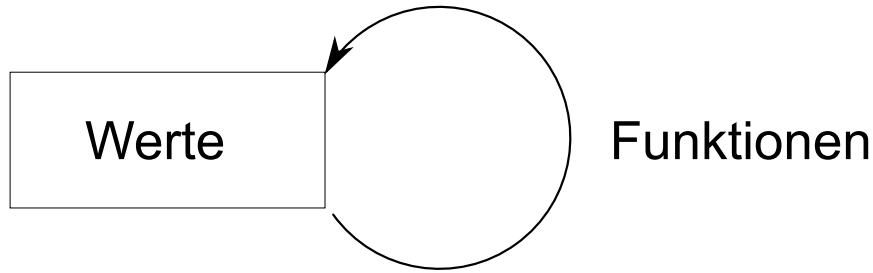
```
f x y =
 let y = M
     f x = N x
 in  if g then P y
          else Q
```

- 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!

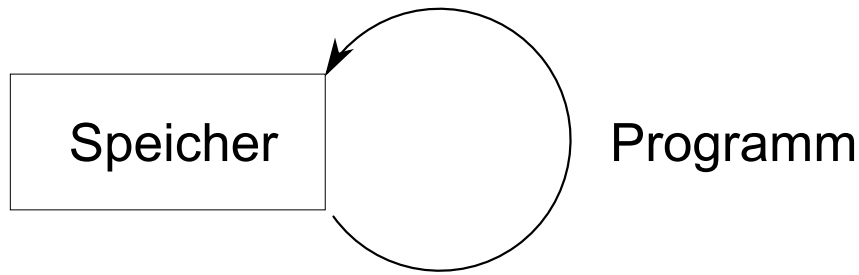
Datentypen als Modellierungskonstrukt

Programme manipulieren ein Modell (der Umwelt)

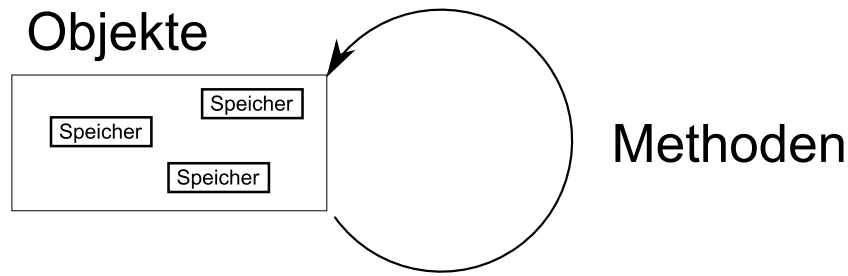
- Funktionale Sicht:



- Imperative Sicht:



- Objektorientierte Sicht:



Datentypen, Funktionen und Beweise

- Datentypen konstruieren Werte
- Funktionen definieren Berechnungen
- Berechnungen haben Eigenschaften
- Dualität:

  Datentypkonstruktor $\longleftrightarrow$ Definitionskonstrukt $\longleftrightarrow$ Beweiskonstrukt

Typkonstruktoren

- Aufzählungen
- Produkt
- Rekursion
- Funktionsraum

Aufzählungen

- Aufzählungen: Menge von disjunkten Konstanten

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

$$Mon \neq Tue, Mon \neq Wed, Tue \neq Thu, Wed \neq Sun \ldots$$

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

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

Aufzählung und Fallunterscheidung in Haskell

- Definition

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

  - Implizite Deklaration der Konstruktoren Mon :: 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
```

Aufzählung und Fallunterscheidung in Haskell

- Definition

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

  - Implizite Deklaration der Konstruktoren Mon :: 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
```

Fallunterscheidung in der Funktionsdefinition

- Abkürzende Schreibweise (syntaktischer Zucker):

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

- Damit:

```
weekend :: Days → Bool
weekend Sat = True
weekend Sun = True
weekend _   = False
```

Der einfachste Aufzählungstyp

- Einfachste Aufzählung: Wahrheitswerte

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

  - Genau zwei unterschiedliche Werte

- Definition von Funktionen:

  - Wertetabellen sind explizite Fallunterscheidungen

$\wedge$	*true*	*false*
true	*true*	*false*
false	*false*	*false*

$$\begin{aligned} true \wedge true &= true \\ true \wedge false &= false \\ false \wedge true &= false \\ false \wedge false &= false \end{aligned}$$

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 && div 1 0 ==0 ⇝ False
- if then else als syntaktischer Zucker:

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

Beispiel: Ausschließende Disjunktion

- Mathematische Definiton:

```
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

Produkte

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

$$\begin{aligned} \mathit{Date} &= \{\mathit{Date}\ (n,\ m,\ y) \mid n \in \mathbb{N}, m \in \mathit{Month}, y \in \mathbb{N}\} \\ \mathit{Month} &= \{\mathit{Jan}, \mathit{Feb}, \mathit{Mar}, \ldots\} \end{aligned}$$

- Funktionsdefinition:
  - Konstruktorargumente sind gebundene Variablen

$$\begin{aligned} \mathit{year}(D(n,m,y)) &= y \\ \mathit{day}(D(n,m,y)) &= n \end{aligned}$$

  - Bei der Auswertung wird gebundene Variable durch konkretes Argument ersetzt

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
```

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
```

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
```

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
```

- Berechung des Umfangs:

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

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{aligned} \text{data } T = \quad & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \ldots \\ \mid \quad & C_n\ t_{n,1} \ldots t_{n,k_n} \end{aligned}$$

- Konstruktoren $C_1, \ldots, C_n$ sind disjunkt:
$$C_i\ x_1 \ldots x_n = C_j\ y_1 \ldots y_m \longrightarrow i = j$$
- Konstruktoren sind injektiv:
$$C\ x_1 \ldots x_n = C\ y_1 \ldots y_n \longrightarrow x_i = y_i$$
- Konstruktoren erzeugen den Datentyp:
$$\forall x \in T.\, x = C_i\ y_1 \ldots y_m$$

Diese Eigenschaften machen Fallunterscheidung möglich.

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

Wie beweisen?

- Gleichungsumformung (equational reasoning)
- Fallunterscheidungen
- Induktion
- Wichtig: formale Notation

Ein ganz einfaches Beispiel

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

$$\frac{\textbf{Lemma:}\quad \mathsf{addTwice}\ x\ (y+z) = \mathsf{addTwice}\ (x+y)\ z}{\mathsf{addTwice}\ x\ (y+z)}$$

Ein ganz einfaches Beispiel

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

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

$$
\begin{aligned}
& \text{addTwice}\ x\ (y+z) \\
= \quad & 2*(x+(y+z)) \qquad \text{— Def. addTwice}
\end{aligned}
$$

Ein ganz einfaches Beispiel

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

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

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

Ein ganz einfaches Beispiel

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

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

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

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: $\max\ x\ y - \min\ x\ y = |x - y|$

$\max\ x\ y - \min\ x\ y$

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: $\max\ x\ y - \min\ x\ y = |x - y|$

$\max\ x\ y - \min\ x\ y$

- Fall: $x < y$

$= \quad y - \min\ x\ y \quad$ — Def. max

$= \quad y - x \quad$ — Def. min

$= \quad |x - y| \quad$ — Wenn $x < y$, dann $y - x = |x - y|$

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: $\max\ x\ y - \min\ x\ y = |x - y|$

$\max\ x\ y - \min\ x\ y$

- Fall: $x < y$

$= \quad y - \min\ x\ y \quad$ — Def. max

$= \quad y - x \quad$ — Def. min

$= \quad |x - y| \quad$ — Wenn $x < y$, dann $y - x = |x - y|$

- Fall: $x \geq y$

$= \quad x - \min\ x\ y \quad$ — Def. max

$= \quad x - y \quad$ — Def. min

$= \quad |x - y| \quad$ — Wenn $x \geq y$, dann $x - y = |x - y|$

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: $\max\ x\ y - \min\ x\ y = |x - y|$

$\max\ x\ y - \min\ x\ y$

- Fall: $x < y$

$= \quad y - \min\ x\ y$ — Def. max

$= \quad y - x$ — Def. min

$= \quad |x - y|$ — Wenn $x < y$, dann $y - x = |x - y|$

- Fall: $x \geq y$

$= \quad x - \min\ x\ y$ — Def. max

$= \quad x - y$ — Def. min

$= \quad |x - y|$ — Wenn $x \geq y$, dann $x - y = |x - y|$

$= \quad |x - y|$

$\square$

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

$\longleftrightarrow$

beliebige Genauigkeit,
wachsender Aufwand

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand $\longleftrightarrow$ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

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

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 (div x y)*y + mod x y == x

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

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 Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

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

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