Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2017/2018

(Stand: 20.12.2017)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

Inhaltsverzeichnis

Inhaltsverzeichnis (1)

Inhaltsverzeichnis (2)

Teil II: Grundlagen

Inhaltsverzeichnis (3)

Inhaltsverzeichnis (4)

Inhaltsverzeichnis (5)

Inhaltsverzeichnis (6)

Teil III: Applikative Programmierung

Inhaltsverzeichnis (7)

Inhaltsverzeichnis (8)

Inhaltsverzeichnis (9)

Inhaltsverzeichnis (10)

Teil VI: Weiterführende Konzepte

Inhaltsverzeichnis (11)

Inhaltsverzeichnis (12)

Teil I

Einführung

Kapitel 1

Motivation

Das leere Haskell-Programm

Das leere Haskell-Programm: Mehr als nichts!

...bereits das leere Haskell-Programm bietet

Taschenrechnerfunktionalität:

```
>hugs
Main>:load leeresHaskellProgramm.hs
Main>2+3
5
Main>abs (5-12)
7
Main>sqrt 121
11.0
Main>abs (-5) * 6 + 3 <= 2^3 * (4 + round 3.14)
True
Main>sin 0
0.0
Main>cos 0
1.0
```

Das leere Haskell-Programm: Mehr als nichts!

...und mehr:

```
Main>True && False
False
Main>not (True && False)
True
Main>"Funktionale" ++ " " ++ "Programmierung"
"Funktionale Programmierung"
Main>length "Funktionale Programmierung"
26
Main>[1..12]
[1,2,3,4,5,6,7,8,9,10,11,12]
Main>[1,4..12]
[1,4,7,10]
Main>length [10..20]
11
Main>[ n | n <- [-6..8], mod n 2 == 0 ]
[-6,-4,-2,0,2,4,6,8]
```

Überblick

Funktionale Programmierung, funktionale Programmierung in Haskell

Anmerkung: Einige Begriffe werden in diesem Kapitel im Vorgriff angerissen und erst im Lauf der Vorlesung genau geklärt!

Kapitel 1.1

Ein Beispiel sagt (oft) mehr als 1000 Worte

Kapitel 1.1.1

Zehn Beispiele

Zehn Beispiele

1. *Hello, World!*
2. Fakultätsfunktion
3. Das Sieb des Eratosthenes
4. Binomialkoeffizienten
5. Umkehren einer Zeichenreihe
6. Reißverschlussfunktion
7. Addition
8. Map-Funktion
9. Euklidischer Algorithmus
10. Gerade/ungerade-Test

1) Hello, World!

```
main = putStrLn "Hello, World!"
```

...ein Beispiel für ein Programm mit Ein-/Ausgabeoperation.

Nicht selbsterklärend: Die Deklaration von `putStrLn`:

```
putStrLn :: String -> IO ()
putStrLn "Hello, World!"
```

Allerdings: Auch die Java-Entsprechung

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello, World!"); } }
```

...bedarf einer weiter ausholenden Erläuterung.

2) Fakultätsfunktion (1)

$$! : \mathbb{N} \to \mathbb{N}$$

$$\forall\, n \in \mathbb{N}.\ n\,! \;=\; \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{sonst} \end{cases}$$

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n - 1)
```

...ein Beispiel für eine rekursive Funktionsdefinition.

Aufrufe:

```
fac 0 ->> 1    fac 3 ->> 6      fac 6  ->> 720
fac 1 ->> 1    fac 5 ->> 120    fac 10 ->> 3.628.800
```

Lies: "Die Auswertung des Ausdrucks/Aufrufs `fac 5` liefert den Wert 120; der Ausdruck/Aufruf `fac 5` hat den Wert 120."

2) Fakultätsfunktion (2)

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Funktionale Programmierung mag es kurz und knackig, prägnant und konzis, ohne kryptisch zu sein. Auch Haskell hat hierfür ein Angebot.

Alternative Schreibweise:

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1                       (| für (oder) wenn)
 | otherwise = n * fac (n - 1)    (otherwise ->> True)

fac :: Integer -> Integer       (Diese Variante nur zur
fac n                                  Illustration von |)
 | n == 0 || n == 1 = 1               ((||) logisches oder)
 | n == 2           = 2
 | otherwise        = n * fac (n - 1)
```

2) Fakultätsfunktion (3)

Eine weitere Schreibweise, musterbasiert:

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)
```

Alternative Implementierungen:

```
fac :: Integer -> Integer
fac n = foldl (*) 1 [1..n]      ([1..3] ->> [1,2,3],
                                  [1..0] ->> [],
           foldl (*) 1 [1..0] ->> foldl (*) 1 [] ->> 1)
```

```
fac :: Integer -> Integer
fac n = product [1..n]      (product = foldl (*) 1 )
```

2) Fakultätsfunktion (4)

Eine einfache Form der Fehlerbehandlung (Panikmodus):

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1
 | n > 0     = n * fac (n - 1)
 | otherwise = error "fac: Unzulässiges Argument!"
```

Aufrufe:

```
fac 10   ->> 3.628.800
fac 5    ->> 120
fac 0    ->> 1
fac (-1) ->> "fac: Unzulässiges Argument!"
fac (-5) ->> "fac: Unzulässiges Argument!"
...
```

Funktional vs. Imperativ: Kurzer Exkurs (1)

Vergleiche folgende funktionale und imperative Implementierungen der Fakultätsfunktion:

Funktional, hier in Haskell:

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n-1)
```

Imperativ, hier in Pascal:

```
FUNCTION fac (n: integer): integer;
BEGIN
  IF n=0 THEN fac := 1 ELSE fac := n*fac(n-1)
END;
```

Beachte: Trotz der äußerlichen Ähnlichkeit sind die funktionale und imperative Fallunterscheidung, die in beiden Fällen die Fakultätsfunktion definieren, konzeptuell äußerst verschieden!

Funktional vs. Imperativ: Kurzer Exkurs (2)

Die Fallunterscheidung “if-then-else” im Vergleich:

Funktional:

```
fac :: Integer -> Integer
```

$$\texttt{fac n = }\underbrace{\texttt{if }\underbrace{\texttt{n == 0}}_{\textit{Ausdruck}}\texttt{ then }\underbrace{\texttt{1}}_{\textit{Ausdruck}}\texttt{ else }\underbrace{\texttt{n * fac (n-1)}}_{\textit{Ausdruck}}}_{\textit{Ausdruck}}$$

Imperativ:

```
FUNCTION fac (n: integer): integer;
```

$$\texttt{BEGIN }\underbrace{\texttt{IF }\underbrace{\texttt{n=0}}_{\textit{Ausdruck}}\texttt{ THEN }\underbrace{\texttt{fac := }\underbrace{\texttt{1}}_{\textit{Ausdruck}}}_{\textit{Anweisung}}\texttt{ ELSE }\underbrace{\texttt{fac := }\underbrace{\texttt{n*fac(n-1)}}_{\textit{Ausdruck}}}_{\textit{Anweisung}}}_{\textit{Anweisung}}\texttt{ END;}$$

Inhalt
Kap. 1
1.1
1.1.1
1.1.2
1.2
1.3
1.4
Kap. 2
Kap. 3
Kap. 4
Kap. 5
Kap. 6
Kap. 7
Kap. 8
Kap. 9
Kap. 10
Kap. 11
Kap. 12
Kap. 13
Kap. 14

Funktional vs. Imperativ: Kurzer Exkurs (3)

Die Fallunterscheidung “if-then-else”:

- Funktional: Die Fallunterscheidung ist ein Ausdruck. Ihre Bedeutung (Semantik) ist ein Wert.
- Imperativ: Die Fallunterscheidung ist eine Anweisung. Ihre Bedeutung (Semantik) ist eine Zustandstransformation, eine Belegung von Variablen mit (neuen) Werten.

Dieser Unterschied in Konzept u. Bedeutung ist fundamental.

- “if-then-else” funktional $\neq$ “if-then-else” imperativ

Es ist wichtig, sich diesen Unterschied klarzumachen.

3) Das Sieb des Eratosthenes (276-194 v.Chr.)

...zur Berechnung der unendlichen Folge der Primzahlen:

1. Schreibe alle natürlichen Zahlen ab 2 hintereinander auf.
2. Die kleinste nicht gestrichene Zahl in dieser Folge ist eine Primzahl. Streiche alle Vielfachen dieser Zahl.
3. Wiederhole Schritt 2 mit der kleinsten jeweils noch nicht gestrichenen Zahl.

```
Nach Schritt 1:
 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17...
Nach Schritt 2 für "2":
 2  3     5     7     9     11    13    15    17...
Nach Schritt 2 für "3":
 2  3     5     7           11    13          17...
usw.
```

3) Das Sieb des Eratosthenes (2)

$$\texttt{primes :: }\underbrace{\texttt{[Integer]}}_{\textit{Zahlenstromtyp}}$$

(primes, der (Prim-) Zahlenstrom als Integer-Liste)

$$\texttt{primes = }\underbrace{\texttt{sieve [2..]}}_{\textit{Strom der nat. Zahlen ab 2}}$$

(leistet Schritt 1)

$$\texttt{sieve :: }\underbrace{\texttt{[Integer]}}_{\textit{Argumentstromtyp}}\texttt{ -> }\underbrace{\texttt{[Integer]}}_{\textit{Resultatstromtyp}}$$

$$\underbrace{\texttt{sieve (x:xs)}}_{\textit{Argumentstrom}}\texttt{ = }\underbrace{\texttt{x : sieve [y | y <- xs, mod y x > 0]}}_{\textit{Resultatstrom}}$$

(leistet Schritt 2 für 2, 3, 5, 7, 11, usw.)

...ein Beispiel für die Programmierung mit Strömen.

3) Das Sieb des Eratosthenes (3)

$$\texttt{primes} :: \underbrace{\texttt{[Integer]}}_{\textit{(Prim}-\textit{) Zahlenstromtyp}}$$

$$\texttt{sieve} :: \underbrace{\texttt{[Integer]}}_{\textit{Argumentstromtyp}} \;\texttt{->}\; \underbrace{\texttt{[Integer]}}_{\textit{Resultatstromtyp}}$$

$$\underbrace{\texttt{sieve (x:xs)}}_{\substack{\textit{Strom der nat. Zahlen ab 2}\\ \textit{als Argument}}} = \underbrace{\texttt{x : sieve [y | y <- xs, mod y x > 0]}}_{\substack{\textit{Strom der Primzahlen}\\ \textit{als Resultat}}}$$

$$\underbrace{\texttt{primes}}_{\textit{Strom der Primzahlen}} = \overbrace{\texttt{sieve [2..]}}^{\textit{Strom der nat. Zahlen ab 2}}$$

Aufruf:

```
primes ->> sieve [2..] ->> 2 : sieve [3,5..]
 ->> ... ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...
```

3) Das Sieb des Eratosthenes (4)

Im Überblick und (fast) ohne Farbspiele:

```
primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]
```

Aufrufe:

```
primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...
take 5 primes ->> [2,3,5,7,11]
drop 3 primes ->> [7,11,13,17,19,23,29,31,37,41,...
take 3 (drop 3 primes) ->> [7,11,13]
primes!!0 ->> 2              ((!!) Zugriffsoperator)
primes!!1 ->> 3
primes!!5 ->> 13
```

4) Binomialkoeffizienten (1)

...geben die Anzahl der Kombinationen k-ter Ordnung von n Elementen ohne Wiederholung an:

$$\binom{\cdot}{\cdot} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\forall\, n, k \in \mathbb{N}.\ \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

...ein Beispiel für eine musterbasierte Funktionsdefinition mit hierarchischer Abstützung auf eine andere Funktion ("Hilfsfunktion"), hier die Fakultätsfunktion.

Aufrufe:

```
binom' (49,6) ->> 13.983.816
binom' (45,6) ->>  8.145.060
```

4) Binomialkoeffizienten (2)

Es gilt:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)
  | k==0 || n==k = 1
  | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

...ein Beispiel für eine musterbasierte (kaskaden- oder baumartig-) rekursive Funktionsdefinition.

Aufrufe:

```
binom' (49,6) ->> 13.983.816
binom' (45,6) ->>  8.145.060
```

4) Binomialkoeffizienten (3)

Uncurryfiziert

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

Curryfiziert

```
binom :: Integer -> (Integer -> Integer)
binom n k  = div (fac n) (fac k * fac (n-k))
```

Aufrufe:

```
binom' (49,6) ->> 13.983.816
binom' (45,6) ->>  8.145.060

binom 49 6  ->> 13.983.816
binom 45 6  ->>  8.145.060

binom 49
binom 45 ...sind ebenfalls zulässige Ausdrücke!
```

4) Binomialkoeffizienten (4)

Die Aufrufe

```
binom 49
binom 45
```

...sind gültige Ausdrücke von einem funktionalen Wert:

```
(binom 49) :: Integer -> Integer
(binom 45) :: Integer -> Integer
```

...und repräsentieren die Funktionen "49_über_k" (entsprechend k_aus_49") und "45_über_k" (entsprechend "k_aus_45"):

$$\binom{49}{\cdot} : \mathbb{N} \to \mathbb{N} \qquad \binom{45}{\cdot} : \mathbb{N} \to \mathbb{N}$$

$$\forall\, k \in \mathbb{N}.\ \binom{49}{k} = \frac{49!}{k!(49-k)!} \qquad \forall\, k \in \mathbb{N}.\ \binom{45}{k} = \frac{45!}{k!(45-k)!}$$

4) Binomialkoeffizienten (5)

In der Tat können wir als Funktionen definieren:

```
k_aus_49 :: Integer -> Integer
k_aus_49 k = binom 49 k

k_aus_45 :: Integer -> Integer
k_aus_45 k = binom 45 k
```

...und punktfrei (d.h., argumentlos) noch knapper:

```
k_aus_49 :: Integer -> Integer
k_aus_49 = binom 49

k_aus_45 :: Integer -> Integer
k_aus_45 = binom 45
```

Aufrufe:

```
k_aus_49 6 ->> binom 49 6 ->> 13.983.816
k_aus_45 6 ->> binom 45 6 ->>  8.145.060
```

5) Umkehren einer Zeichenreihe

```
data [a] = [] | a:[a]         (Algebraische Datentyp-
                                   spez. für Listen)
type String = [Char]                     (Typsynonym)
reverse :: String -> String
reverse ""      = ""  (""==[], leere Zeichenreihe)
reverse (c:cs) = (reverse cs) ++ [c]
```

...ein Beispiel für eine Funktion auf Zeichenreihen.

Aufrufe:

```
reverse "" ->> ""
reverse "stressed" ->> "desserts"
reverse "desserts" ->> "stressed"
```

6) Reißverschlussfunktion

...zum 'Verpaaren' zweier Listen zu einer Liste von Paaren:

```
zip :: [a] -> [b] -> [(a,b)]    (a, b Typvariablen)
zip _ []          = []          (_ sog. "wild card")
zip [] _          = []          ([] leere Liste)
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Liste mit Kopf y und Rest ys
: sog. Listenkonstruktor

...ein Beispiel für eine polymorphe Funktion auf Listen.

Aufrufe:

```
zip [2,3,5,7] ['a','b'] ->> [(2,'a'),(3,'b')]
zip [] ["stressed","desserts"] ->> []
zip [1.1,2.2.3.3] ["fun"] ->> [(1.1,"fun")]
```

7) Addition

```
(+) :: Num a => a -> a -> a    (Num sog. Typklasse)
```

...ein Beispiel für eine überladene Funktion.

Aufrufe:

```
(+) 2 3       ->> 5        (+ auf ganzen Z., Präfixop.)
2 + 3         ->> 5          (+ als Infixop. auf g.Z.)
(+) 2.1 1.4   ->> 3.5 (+ auf Gleitkommaz., Präfixop.)
2.1 + 1.4     ->> 3.5          (+ als Infixop. auf Gkz)
(+) 7.81 2    ->> 9.81   (automatische Typanpassung)
```

```
((+) 1) :: Integer -> Integer  (Inkrementfunktion)
```

```
inc :: Integer -> Integer
inc = (+) 1          (vgl. die Funktion "(binom 49)")
```

```
inc' :: Integer -> Integer
inc' = (+1)         ((+1) ein sog. Operatorabschnitt)
```

8) Die map-Funktion

...zur Anwendung einer Funktion auf alle Elemente einer Liste:

```
map :: (a -> b) -> [a] -> [b]        (Fkt. als Arg.)
map _ [] = []
map f (x:xs) = (f x) : map f xs
```

...ein Beispiel für eine Funktion höherer Ordnung, für Funktionen als Bürger erster Klasse (first class citizens).

Aufrufe:

```
map (2*) [1,2,3,4,5] ->> [2,4,6,8,10]
map (\x -> x*x) [1,2,3,4,5] ->> [1,4,9,16,25]
map (>3) [2,3,4,5] ->> [False,False,True,True]
map length ["functional","programming","is","fun"]
                                   ->> [10,11,2,3]
```

9) Der Euklidische Algorithmus (3.Jhdt.v.Chr.)

...zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen m, n mit $m \geq 0$ und $n > 0$:

```
ggt :: Int -> Int -> Int (Ganzz.-Typ, beschränkt)
ggt m n
  | n == 0 = m
  | n > 0  = ggt n (mod m n)

mod :: Int -> Int -> Int
mod m n
  | m < n  = m
  | m >= n = mod (m-n) n
```

...ein Beispiel für ein hierarchisches System von Funktionen.

Aufrufe:

```
ggt 25 15 ->> 5   ggt 48 60 ->> 12   mod 8 3 ->> 2
ggt 28 60 ->> 4   ggt 60 40 ->> 20   mod 9 3 ->> 0
```

10) Gerade/ungerade-Test

```
isEven :: Integer -> Bool
isEven n
 | n == 0 = True
 | n > 0  = isOdd (n-1)

isOdd :: Integer -> Bool
isOdd n
 | n == 0 = False
 | n > 0  = isEven (n-1)
```

...ein Beispiel für ein System wechselweise (bzw. indirekt) rekursiver Funktionen.

Aufrufe:

```
isEven 6 ->> True               isEven 9 ->> False
isOdd 6  ->> False              isOdd 9  ->> True
isEven (-5) Musterfehler        isOdd (-1) Musterfehler
```

Rückblickend – die ersten zehn Beispiele (1)

1. Ein- und Ausgabe
   - *Hello, World!*
2. Rekursion
   - Fakultätsfunktion
3. Stromprogrammierung
   - Das Sieb des Eratosthenes
4. Musterbasierte, curryfizierte und uncurryfizierte Funktionsdefinitionen, partiell ausgewertete Funktionen
   - Binomialkoeffizienten
5. Funktionen auf Zeichenreihen
   - Umkehren einer Zeichenreihe

Rückblickend – die ersten zehn Beispiele (2)

6. Parametrisch polymorphe Funktionen
   - Reißverschlussfunktion
7. Überladene Funktionen
   - Addition
8. Fkt. höherer Ordnung, Fkt. als "Bürger erster Klasse"
   - Map-Funktion
9. Hierarchische Systeme von Funktionen
   - Euklidischer Algorithmus
10. Systeme wechselweise rekursiver Funktionen
    - Gerade/ungerade-Test

Wir halten fest

Funktionale Programme sind

- Systeme (wechselweise) rekursiver Funktionsvorschriften.

Funktionen

- sind zentrales Abstraktionsmittel in funktionalen Programmen (wie Prozeduren/Methoden in prozeduralen/objektorientierten Programmen).

Funktionale Programme

- werten Ausdrücke aus. Das Resultat dieser Auswertung ist ein Wert eines bestimmten Typs. Dieser Wert kann elementar oder funktional sein; er ist die Bedeutung, die Semantik des Ausdrucks.

Kapitel 1.1.2

Programme auswerten, Programme finden

Auswerten einfacher Ausdrücke (1)

Der Ausdruck `(15*7 + 12) * (7 + 15*12)` hat den Wert 21.879; seine Semantik ist der Wert 21.879.

```
    (15*7 + 12) * (7 + 15*12)
->> (105 + 12) * (7 + 180)
->> 117 * 187
->> 21.879
```

Auch andere Auswertungsreihenfolgen sind möglich, z.B.:

```
    (15*7 + 12) * (7 + 15*12)
->> (105 + 12) * (7 + 180)
->> 105*7 + 105*180 +12*7 + 12*180
->> 735 + 18.900 + 84 + 2.160
->> 21.879
```

Auswerten einfacher Ausdrücke (2)

```
    (15*7 + 12) * (7 + 15*12)
->> (105 + 12) * (7 + 180)
->> 117 * (7 + 180)
->> 117*7 + 117*180
->> 819 + 21.060
->> 21.879
```

...und viele mehr. Stets ist das Ergebnis gleich! Siehe Kapitel 12.3, Church-Rosser-Theoreme.

Die einzelnen Vereinfachungs-, Rechenschritte nennen wir

- Simplifikationen.

Auswerten funktionaler Ausdrücke (1)

Der Ausdruck `fac 2` hat den Wert 2; seine Semantik ist der Wert 2.

Eine erste Auswertungsreihenfolge:

```
        fac 2
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
(S) ->> if False then 1 else (2 * fac (2-1))
(S) ->> 2 * fac (2-1)
(S) ->> 2 * fac 1
(E) ->> 2 * (if 1 == 0 then 1 else (1 * fac (1-1)))
(S) ->> 2 * (if False then 1 else (1 * fac (1-1)))
(S) ->> 2 * (1 * fac (1-1))
(S) ->> 2 * (1 * fac 0)
(E) ->> 2 * (1 * (if 0 == 0 then 1 else (0 * fac (0-1))))
(S) ->> 2 * (1 * (if True then 1 else (0 * fac (0-1))))
(S) ->> 2 * (1 * (1))
(S) ->> 2 * (1 * 1)
(S) ->> 2 * 1
(S) ->> 2
```

Auswerten funktionaler Ausdrücke (2)

Eine zweite Auswertungsreihenfolge:

```
          fac 2
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
(S) ->> if False then 1 else (2 * fac (2-1))
(S) ->> 2 * fac (2-1)
(E) ->> 2 * (if (2-1) == 0 then 1
                 else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * (if 1 == 0 then 1
                 else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * (if False then 1
                 else ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * ((2-1) * fac ((2-1)-1)))
(S) ->> 2 * (1 * fac ((2-1)-1)))
(E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1
                      else (((2-1)-1) * fac ((2-1)-1))))
(S) ->> 2 * (1 * (if (1-1) == 0 then 1
                      else (((2-1)-1) * fac ((2-1)-1))))
```

Auswerten funktionaler Ausdrücke (3)

```
(S) ->> 2 * (1 * (if 0 == 0 then 1
                  else (((2-1)-1) * fac ((2-1)-1))))
(S) ->> 2 * (1 * (if True then 1
                  else (((2-1)-1) * fac ((2-1)-1))))
(S) ->> 2 * (1 * 1)
(S) ->> 2 * 1
(S) ->> 2
```

Wir bezeichnen die mit

- (E) markierten Schritte als Expansionsschritte.
- (S) markierten Schritte als Simplifikationsschritte.

Auswerten funktionaler Ausdrücke (4)

Die beiden Auswertungsreihenfolgen sind Beispiele

- applikativer (unverzüglicher) (1. Ausw.folge, z.B. in ML)
- normaler (verzögerter) (2. Ausw.folge, z.B. in Haskell)

Auswertung.

Applikative Auswertung des Aufrufs natSum 2

```
          natSum 2
(E) ->> if 2 == 0 then 0 else (natSum (2-1)) + 2
(S) ->> if False then 0 else (natSum (2-1)) + 2
(S) ->> (natSum (2-1)) + 2
(S) ->> (natSum 1) + 2
(E) ->> (if 1 == 0 then 0 else ((natSum (1-1)) +1)) + 2
(S) ->> (if False then 0 else ((natSum (1-1)) + 1)) + 2
(S) ->> ((natSum (1-1)) + 1) + 2
(S) ->> ((natSum 0) + 1) + 2
(E) ->> ((if 0 == 0 then 0 else (natSum (0-1)) + 0) + 1) + 2
(E) ->> ((if True then 0 else (natSum (0-1)) + 0) + 1) + 2
(S) ->> ((0) + 1) + 2
(S) ->> (0 + 1) + 2
(S) ->> 1 + 2
(S) ->> 3
```

Normale Auswertung des Aufrufs `natSum 2`

```
         natSum 2
(E) ->> if 2 == 0 then 0 else (natSum (2-1)) + 2
(S) ->> if False then 0 else (natSum (2-1)) + 2
(S) ->> (natSum (2-1)) + 2
(E) ->> if (2-1) == 0 then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> if 1 == 0 then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> if False then 0 else (natSum ((2-1)-1)) + (2-1) + 2
(S) ->> (natSum ((2-1)-1)) + (2-1) + 2
(E) ->> ...
...
(S) ->> 3
```

Übungsaufgabe 1.1.2.1

Vervollständige die normale Auswertung des Aufrufs `natSum 2`.

‘Finden’ rekursiver Formulierungen (1)

...am Beispiel der Fakultätsfunktion:

```
fac n = n*(n-1)*...*6*5*4*3*2*1*1
```

Von der Lösung erwarten wir:

```
fac 0 = 1 ->> 1
fac 1 = 1*1 ->> 1
fac 2 = 2*1*1 ->> 2
fac 3 = 3*2*1*1 ->> 6
fac 4 = 4*3*2*1*1 ->> 24
fac 5 = 5*4*3*2*1*1 ->> 120
fac 6 = 6*5*4*3*2*1*1 ->> 720
...

fac n = n*(n-1)*...*6*5*4*3*2*1*1 ->> n!
```

'Finden' rekursiver Formulierungen (2)

Beobachtung:

```
fac 0 = 1                ->> 1
fac 1 = 1 * fac 0        ->> 1
fac 2 = 2 * fac 1        ->> 2
fac 3 = 3 * fac 2        ->> 6
fac 4 = 4 * fac 3        ->> 24
fac 5 = 5 * fac 4        ->> 120
fac 6 = 6 * fac 5        ->> 720
...

fac n = n * fac (n-1)  ->> n!
```

‘Finden’ rekursiver Formulierungen (3)

Wir erkennen:

- Ein Regelfall: `fac n = n * fac (n-1)`
- Ein Sonderfall: `fac 0 = 1`

Wir führen beide Fälle zusammen und erhalten:

```
fac n =
 | n == 0    = 1
 | otherwise = n * fac (n-1)
```

‘Finden’ rekursiver Formulierungen (4)

...am Beispiel der Berechnung von 0+1+2+3...+n:

```
natSum n = 0+1+2+3+4+5+6+...+(n-1)+n
```

Von der Lösung erwarten wir:

```
natSum 0 = 0 ->> 0
natSum 1 = 0+1 ->> 1
natSum 2 = 0+1+2 ->> 3
natSum 3 = 0+1+2+3 ->> 6
natSum 4 = 0+1+2+3+4 ->> 10
natSum 5 = 0+1+2+3+4+5 ->> 15
natSum 6 = 0+1+2+3+4+5+6 ->> 21
...
natSum n = 0+1+2+3+4+5+6+...+(n-1)+n
```

‘Finden’ rekursiver Formulierungen (5)

Beobachtung:

```
natSum 0 =                0 ->>  0
natSum 1 = (natSum 0) + 1 ->>  1
natSum 2 = (natSum 1) + 2 ->>  3
natSum 3 = (natSum 2) + 3 ->>  6
natSum 4 = (natSum 3) + 4 ->> 10
natSum 5 = (natSum 4) + 5 ->> 15
natSum 6 = (natSum 5) + 6 ->> 21
...

natSum n = (natSum n-1) + n
```

‘Finden’ rekursiver Formulierungen (6)

Wir erkennen:

- Ein Regelfall: `natSum n = (natSum (n-1)) + n`
- Ein Sonderfall: `natSum 0 = 0`

Wir führen beide Fälle zusammen und erhalten:

```
natSum n
 | n == 0    = 0
 | otherwise = natSum (n-1) + n
```

Kapitel 1.2

Warum funktionale Programmierung?
Warum mit Haskell?

Kapitel 1.2.1

Warum funktionale Programmierung?

Die Frage von John W. Backus

"Can programming be liberated from the von Neumann style?"

John W. Backus
Turing Award Preisträger 1978

John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8): 613-641, 1978.

Programmierparadigmen

...vielfältig und zahlreich, darunter:

- **Imperativ**
  - Prozedural (Pascal, Modula, C,...)
  - Objektorientiert (Smalltalk, Java, C++, Eiffel,...)
  - Parallel, datenparallel, verteilt (HPF, Ada, MesaF,...)
- **Deklarativ**
  - **Funktional** (Lisp, ML, Miranda, Haskell, Gofer,...)
  - Logisch (Prolog, Datalog, Gödel,...)
  - Bedingt (constraint prog.) (Oz, Curry, Bertrand,...)
- Mischformen
  - Funktional-logisch (Curry, POPLOG, TOY, Mercury,...),
  - Funktional-objektorientiert (Haskell++, O'Haskell, Scala, OCaml,...)
  - ...
- ...
- Graphisch
  - Graphische Programmiersprachen (Forms/3, FAR,...)

Evolution von Paradigmen und Sprachen (1)

...gekennzeichnet durch die schrittweise Einführung von Abstraktionen mit dem Ziel, Einzelheiten der zugrundeliegenden Rechenmaschine und Programmausführung immer mehr zu verbergen:

- Assembler-Sprachen führen mnemo-technische Instruktionsbezeichner und symbolische Marken ein, um Maschinenbefehle und Programm- und Datenspeicheradressen zu verbergen.
- FORTRAN führt Felder (engl. arrays) und Ausdrücke in mathematisch-üblicher Schreibweise ein, um Register zu verbergen.
- ALGOL-ähnliche Sprachen führen strukturierte Kontrollanweisungen ein, um Sprungbefehle und Sprungmarken zu verbergen (*"goto considered harmful"*).

Evolution von Paradigmen und Sprachen (2)

- Objektorientierte Sprachen führen Sichtbarkeitsebenen und Kapselungen ein, um die Datendarstellung und Speicherverwaltung zu verbergen.
- Deklarative Sprachen, am bekanntesten funktionale und logische Sprachen, verbergen die Auswertungsreihenfolge und verzichten dafür auf Kontrollanweisungen. Reine Sprachen verzichten zusätzlich auf Zuweisungen, um Seiteneffekte auszuschließen.

  In deklarativen Sprachen verschiebt sich dadurch

  - die Programmieraufgabe von der Festlegung der Rechenschritte zur Strukturierung der Anwendungsdaten und Beziehungen der Programmbestandteile.

  Deklarative Sprachen ähneln hierin formalen Spezifikationssprachen, sind aber ausführbar.

Abgrenzung funktionaler u. logischer Sprachen

Funktionale Sprachen

- beruhen auf dem mathematischen Funktionsbegriff.
- Programe sind Systeme von Funktionen, die über Gleichungen, Fallunterscheidungen und Rekursion definiert sind und auf (strukturierten) Daten arbeiten.
- bieten effiziente, anforderungsgetriebene Auswertungsstrategien, die auch die Arbeit mit (potentiell) unendlichen Strukturen unterstützen.

Logische Sprachen

- beruhen auf Prädikatenlogik.
- Programme sind Systeme von Prädikaten, die durch eingeschränkte Formen logischer Formeln, z.B., Horn-Formeln (Implikation), definiert sind.
- bieten Nichtdeterminismus und Prädikate mit mehreren Eingabe-/Ausgabemodi zur Wiederverwendung von Code.

Ziel all dieser Abstraktionen

...maßgebliche Beiträge zur Überwindung der sog. Software-krise zu leisten hin zu einer

- ingenieurmäßigen Software-Entwicklung ("in time, in functionality, in budget")
- verifiziert, wartbar, erweiterbar, etc.

indem dem Programmierer ein

- angemessen(er)es Abstraktionsniveau zur Formulierung, Modellierung und Lösung von Problemen

zur Verfügung gestellt wird.

Prozedural vs. funktional: Ein Vergleich

Gegeben eine Aufgabe A, gesucht eine Lösung L für A.

Prozedural: Lösungsablauf typischerweise in 2 Schritten:

1. Ersinne ein algorithmisches Verfahren V zur Berechnung der Lösung L von A.
2. Codiere V als Folge von Anweisungen (Kommandos, Instruktionen) für den Rechner.

Beachte:

- Schritt 2 erfordert hier zwingend, den Speicher explizit anzusprechen und zu verwalten (Allokation, Manipulation, Deallokation von Speicherzellen für Daten).

Zur Illustration ein einfaches Beispiel (1)

Aufgabe: *"Liefere alle Einträge eines ganzzahligen Feldes mit einem Wert von höchstens 10."*

Hier eine typische prozedurale Lösung, hier in Pascal (Argument in Feldvariable a, Resultat in Feldvariable b):

```
PROGRAM filter (input,output);
VAR a, b: ARRAY [1..maxLength] OF integer;
BEGIN
 (* Code zur Initialisierung von Feldvariable a:
    ... *)
 j := 1;
 FOR i:=1 TO maxLength DO
     IF a[i] <= 10 THEN
        BEGIN b[j] := a[i]; j := j+1 END
END.
```

Beachte: Der Speicher wird explizit adressiert und manipuliert. Zusätzlich ist "Overhead" Code erforderlich.

Zur Illustration ein einfaches Beispiel (2)

...zum Vergleich hier eine typische funktionale Lösung, hier in Haskell:

```
a = [2,5..21]        ( [2,5..21] = [2,5,8,11,14,17,20])
b = [n | n <- a, n <= 10]
```

Beachte: Keine Speicheradressierung, -manipulation oder -verwaltung zur Berechnung von `b` erforderlich: `b ->> [2,5,8]`. Kein "Overhead" Code. Sogar noch knapper möglich:

```
b = [n | n <- [2,5..21], n <= 10]
```

Vergleiche die funktionale und mathematische Beschreibung

- `[ n | n <- a , n <= 10 ]`
- $\{ \; n \; | \; n \in \; a \wedge n \leq \; 10 \; \}$

unter dem Anspruch funktionaler Programmierung

- *"...etwas von der Eleganz der Mathematik in die Programmierung zu bringen!"*

Essenz deklarativer Programmierung (1)

...speziell auch funktionaler Programmierung:

- Das **"was"** in den Vordergrund der Programmierung zu stellen anstatt des **"wie"**!

Deklarativ – Beschreibe, **was** wir bekommen möchten:

```
b = [n | n <- [2,5..21], n <= 10]
```

Prozedural – Beschreibe, **wie** wir es bekommen möchten:

```
VAR a, b: ARRAY [1..maxLength] OF integer;
 (* Code zur Initialisierung von Feldvariable a:
    ... *)
 j := 1;
 FOR i:=1 TO maxLength DO
     IF a[i] <= 10 THEN
        BEGIN b[j] := a[i]; j := j+1 END
```

Essenz deklarativer Programmierung (2)

Automatische Listengenerierung mittels Listenkomprehension (engl. list comprehension) wie im Ausdruck

- `[n | n <- a, n <= 10]` (vgl. $\{n \mid n \in a \land n \leq 10\}$)

...ist hierfür ein wichtiges, nützliches und typisches sprachliches Konstrukt funktionaler Programmiersprachen.

Noch nicht überzeugt?

Betrachte eine komplexere Aufgabe, Sortieren.

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösungsverfahren: Das "Teile und herrsche"-Sortierverfahren Quicksort von Sir Tony Hoare (1961):

- *Teile:* Wähle ein Element l aus L und partitioniere L in zwei (möglicherweise leere) Teillisten L_1 und L_2, so dass alle Elemente von L_1 (L_2) kleiner oder gleich (größer) l sind.
- *Herrsche:* Sortiere L_1 und L_2 mittels des Quicksort-Verfahrens (d.h. mittels rekursiver Aufrufe von Quicksort).
- *Kombiniere:* Bestimme Gesamtsortierung durch Zusammenführen der Teilsortierungen (hier trivial: konkateniere die sortierten Teillisten zur sortierten Gesamtliste).

Quicksort, prozedural

...eine typische prozedurale Realisierung, hier in Pseudocode:

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition (L,low,high)
         quickSort (L,low,splitInd-1)
         quickSort (L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i = low+1 to high do
    if L[i] <= l then left = left+1
                      swap (L[i],L[left]) fi od
  swap (L[low],L[left])
  return left
```

Aufruf: `quickSort(L,1,length(L))`, wobei L die zu sortierende Liste ist, z.B. `L=[4,2,3,4,1,9,3,3]`.

Quicksort, funktional

...zum Vergleich hier eine typische funktionale Realisierung, hier in Haskell:

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                   ++ [n]
                   ++ quickSort [m | m <- ns, m > n]
```

Aufrufe:

```
quickSort [] ->> []
quickSort [4,1,7,3,9] ->> [1,3,4,7,9]
quickSort [4,2,3,4,1,9,3,3] ->> [1,2,3,3,3,4,4,9]
```

Imperative/Funktionale Programmierung (1)

Imperativ:

- Unterscheidung von Ausdrücken und Anweisungen.
- Ausdrücke liefern Werte; Anweisungen bewirken Zustandsänderungen (Seiteneffekte).
- Programmausführung ist die Abarbeitung von Anweisungen; dabei müssen auch Ausdrücke ausgewertet werden.
- Kontrollflussspezifikation mittels spezieller Anweisungen (Sequentielle Komposition, Fallunterscheidung, Schleifen, etc.)
- Variablen sind Verweise auf Speicherplätze. Ihre Werte können im Verlauf der Programmausführung geändert werden.
- Die bewirkte Zustandsänderung ist die Bedeutung des Programms.

Imperative/Funktionale Programmierung (2)

Funktional:

- Keine Anweisungen, ausschließlich Ausdrücke.
- Ausdrücke liefern Werte. Zustandsänderungen (und damit Seiteneffekte) gibt es nicht.
- Programmausführung ist Auswertung eines Ausdrucks. Sein Wert ist Ergebnis und Bedeutung des Programms.
- Keine Kontrollflussspezifikation; allein Datenabhängigkeiten steuern die Auswertung(sreihenfolge).
- Variablen sind an Ausdrücke gebunden. Einmal ausgewertet, ist eine Variable an einen einzelnen Wert gebunden; ein späteres Überschreiben oder Neubelegen ist nicht möglich.

Stärken und Vorteile fkt. Programmierung

- Einfach(er) zu erlernen
  ...da wenige(r) Grundkonzepte (vor allem keinerlei (Maschinen-) Instruktionen; insbesondere keine Zuweisungen, keine Schleifen, keine Sprünge)
- Höhere Produktivität
  ...da Programme signifikant kürzer als funktional vergleichbare imperative Programme sind (Faktor 5 bis 10)
- Höhere Zuverlässigkeit
  ...da Korrektheitsüberlegungen/-beweise einfach(er) (math. Fundierung, keine durchscheinende Maschine)

Schwächen und Nachteile fkt. Programmierung

- Geringe(re) Performanz
  Aber: Große Fortschritte sind gemacht (Performanz oft vergleichbar mit entsprechenden C-Implementierungen); Korrektheit zudem vorrangig gegenüber Geschwindigkeit; einfache(re) Parallelisierbarkeit fkt. Programme.

- Manchmal unangemessen, oft für inhärent zustandsbasierte Anwendungen, zur GUI-Programmierung
  Aber: Eignung einer Methode/Technologie/Programmierstils für einen Anwendungsfall ist stets zu untersuchen und überprüfen; dies ist kein Spezifikum fkt. Programmierung.

  Außerdem: Unterstützung zustandsbehafteter Programmierung in vielen funktionalen Programmiersprachen durch spezielle Mechanismen. In Haskell etwa durch das Monadenkonzept (siehe LVA 185.A05 Fortgeschrittene funktionale Programmierung).

Somit: Häufig vorgebrachte Schwächen und Nachteile fkt. Programmierung (oft) nur vermeintlich und vorurteilsbehaftet.

Einsatzfelder funktionaler Programmierung

...mittlerweile "überall":

- Curt J. Simpson. Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- Jerzy Karczmarczuk. Scientific Computation and Functional Programming. Computing in Science and Engineering 1(3):64-72, 1999.
- Bryan O'Sullivan, John Goerzen, Don Stewart. Real World Haskell. O'Reilly, 2008.
- Yaron Minsky. OCaml for the Masses. Communications of the ACM, 54(11):53-58, 2011.
- Haskell in Industry and Open Source: `www.haskell.org/haskellwiki/Haskell_in_industry`

Kapitel 1.2.2

Warum funktionale Programmierung mit Haskell?

Funktionale Programmierprachen

...vielfältig und zahlreich, z.B.:

- λ-Kalkül (späte 1930er Jahre, Alonzo Church, Stephen Kleene)
- Lisp (frühe 1960er Jahre, John McCarthy)
- ML, SML (Mitte der 1970er Jahre, Michael Gordon, Robin Milner)
- Hope (um 1980, Rod Burstall, David McQuee
- Miranda (um 1980, David Turner)
- OPAL (Mitte der 1980er Jahre, Peter Pepper et al.)
- Haskell (späte 1980er Jahre, Paul Hudak, Philip Wadler et al.)
- Gofer (frühe 1990er Jahre, Mark Jones)
- ...

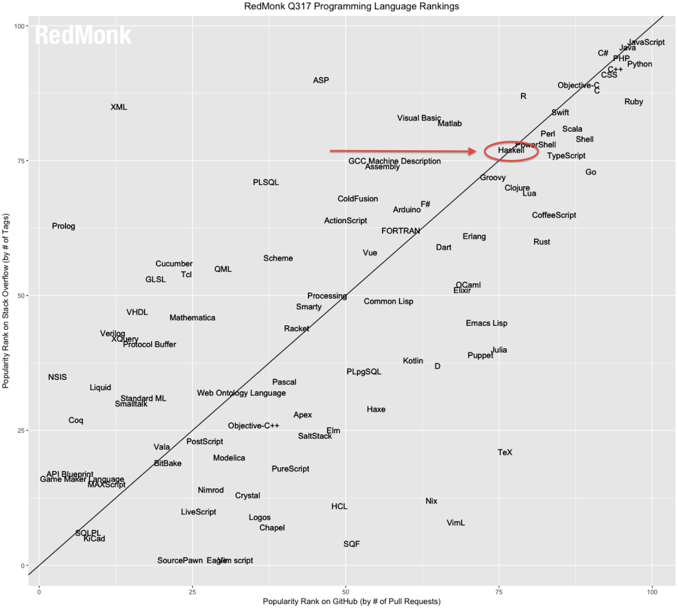
Warum also nicht Haskell?

Haskell ist

- eine fortgeschrittene moderne funktionale Sprache
  - starke Typisierung
  - verzögerte Auswertung (lazy evaluation)
  - Funktionen höherer Ordnung/Funktionale
  - Polymorphie/Generizität
  - Musterpassung (pattern matching)
  - Datenabstraktion (abstrakte Datentypen)
  - Modularisierung (für Programmierung im Großen)
  - ...
- eine Sprache für "realistische (real world)" Probleme
  - mächtige Bibliotheken
  - Schnittstellen zu anderen Sprachen, z.B. zu C
  - ...

In Summe: Haskell ist reich; zugleich ist es eine gute Lehrsprache; auch dank des Interpretierers Hugs! Und Haskell ist mehr als das!

RedMonk Jun'17 Programming Lang. Ranking



RedMonk Programming Language Rankings

Rg	Jan.'15	Rg	Jun'15	Rg	Jan'16	Rg	Jun'16	Rg	Jan'17	Rg	Jun'17
1	JS	1	JS	1	JS	1	JS	1	JS	1	JS
2	Java	2	Java	2	Java	2	Java	2	Java	2	Java
3	PHP	3	PHP	3	PHP	3	PHP	3	Python	3	PHP
4	Python	4	Python	4	Python	4	Python	4	PHP	4	Python
5	C#	5	C#	5	C#	5	C#	5	C#	5	C#
6	C++	5	C++	5	C++	5	C++	5	C++	5	C++
7	Ruby	5	Ruby	5	Ruby	5	Ruby	7	CSS	5	Ruby
8	CSS	8	CSS	8	CSS	8	CSS	7	Ruby	8	CSS
9	C	9	C	9	C	9	C	9	C	9	C
10	Obj-C	10	Obj-C	10	Ob-C	10	Obj-C	10	Obj-C	10	Obj-C
11	Perl	11	Perl	11	Shell	11	Shell	11	Scala	11	Shell
12	Shell	11	Shell	12	Perl	12	R	11	Shell	12	R
13	R	13	R	13	R	13	Perl	11	Swift	13	Perl
14	Scala	14	Scala	14	Scala	14	Scala	14	R	14	Scala
15	Haskell	15	Go	15	Go	15	Go	15	Go	15	Go
16	Matlab	15	Haskell	15	Haskell	16	Haskell	15	Perl	16	Haskell
17	Go	17	Matlab	17	Swift	17	Swift	17	TS	17	Swift
18	VB	18	Swift	18	Matlab	18	Matlab	18	PS	18	Matlab
19	Clojure	19	Groovy	19	Clojure	19	VB	19	Haskell	19	VB
20	Groovy	19	VB	19	Groovy	20	Closure	20	Clojure	20	Closure
				19	VB	20	Groovy	20	CS		20 Groovy
								20	Lua		
								20	Matlab		

Abkürzungen:

JS	JavaScript	TS	TypeScript
Obj-C	Objective-C	PS	PowerShell
VB	Visual Basic	CS	CoffeeScript

URL: `http://redmonk.com`

Steckbrief “Funktionale Programmierung”

Grundlage:	Lambda- (λ-) Kalkül; Basis formaler Berechenbarkeitsmodelle
Abstraktionsprinzip:	Funktionen (höherer Ordnung)
Charakt. Eigenschaft:	Referentielle Transparenz
Historische und aktuelle Bedeutung:	Basis vieler Programmiersprachen; praktische Ausprägung auf dem λ-Kalkül basierender Berechenbarkeitsmodelle
Anwendungsbereiche:	Theoretische Informatik, Künstliche Intelligenz (Expertensysteme), Experimentelle Software/Prototypen, Programmierunterricht,..., Software-Lsg. industriellen Maßstabs
Programmiersprachen:	Lisp, ML, Miranda, Haskell,...

Steckbrief "Haskell"

Benannt nach:	Haskell B. Curry (1900-1982) `www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html`
Paradigma:	Rein funktionale Programmierung
Eigenschaften:	Lazy evaluation, pattern matching
Typsicherheit:	Stark typisiert, Typinferenz, modernes polymorphes Typsystem
Syntax:	Komprimiert, kompakt, intuitiv
Informationen:	`http://haskell.org` `http://haskell.org/tutorial/`
Interpretierer:	Hugs (`haskell.org/hugs/`)
Compiler:	Glasgow Haskell Compiler (GHC)

Haskell-Programme

...gibt es in zwei sich konzeptuell und notationell unterscheidenden Varianten.

Als sog.

- (Gewöhnliches) Haskell-Skript
  ...alles, was nicht notationell als Kommentar ausgezeichnet ist, wird als Programmtext betrachtet.

  Konvention: `.hs` als Dateiendung

- Literates Haskell-Skript (engl. literate Haskell Script)
  ...alles, was nicht notationell als Programmtext ausgezeichnet ist, wird als Kommentar betrachtet.

  Konvention: `.lhs` als Dateiendung

myFirstScript.hs: Gewöhnliches Haskell-Skript

```
{- myFirstScript.hs: Gewöhnliche Skripte erhalten
   konventionsgemäß die Dateiendung .hs -}

-- Fakultätsfunktion
fac :: Integer -> Integer
fac n = if n == 0 then 1 else n * fac (n-1)

-- Binomialkoeffizienten
binom' :: (Integer,Integer) -> Integer
binom' (n,k) = div (fac n) (fac k * fac (n-k))

-- Konstante (0-stellige) Funktion sechsAus45
sechsAus45 :: Integer
sechsAus45 = (fac 45) `div` (fac 6 * fac (45-6))
```

myFirstLitScript.lhs: Literates Haskell-Skript

myFirstLitScript.lhs: Literate Skripte erhalten konventionsgemäß die Dateiendung .lhs

Fakultätsfunktion

```
> fac :: Integer -> Integer
> fac n = if n == 0 then 1 else n * fac(n-1)
```

Binomialkoeffizienten

```
> binom' :: (Integer,Integer) -> Integer
> binom' (n,k) = div (fac n) (fac k * fac (n-k))
```

Konstante (0-stellige) Funktion sechsAus45

```
> sechsAus45 :: Integer
> sechsAus45 = (fac 45) 'div' (fac 6 * fac (45-6))
```

Kommentare in Haskell-Programmen

Kommentare in

- (gewöhnlichem) Haskell-Skript
  - Einzeilig: Alles nach -- bis zum Rest der Zeile
  - Mehrzeilig: Alles zwischen {- und -}
- literatem Haskell-Skript
  - Jede nicht durch > eingeleitete Zeile
    (Beachte: Kommentar- und Codezeilen müssen durch mindestens eine Leerzeile getrennt sein.)

Das Haskell-Vokabular

21 Schlüsselwörter, mehr nicht:

```
case  class  data  default  deriving  do  else
if  import  in  infix  infixl infixr  instance
let  module newtype  of  then  type  where
```

Schlüsselwörter haben

- wie in anderen Programmiersprachen eine besondere Bedeutung und dürfen nicht als Identifikatoren für z.B. Funktionen oder Funktionsargumente verwendet werden.

Identifikatoren

Identifikatoren sind

- nichtleere Zeichenfolgen aus Klein- und Großbuchstaben, Ziffern, einfachen Hochkommata ' und Unterstrichen _, die mit einem Buchstaben beginnen.

  Die Verwendung des Identifikators (z.B. als Funktionsname, Typname, etc.) legt fest, ob der erste Buchstabe ein Kleinbuchstabe (z.B. für Funktionsnamen) oder ein Großbuchstabe (z.B. für Typnamen) sein muss.

Module Prelude: Standard Prelude

Die Definitionen

- einiger der in diesem Kapitel beispielhaft betrachteten Rechenvorschriften und vieler weiterer allgemein nützlicher Deklarationen von Typen und Rechenvorschriften finden sich im vordefinierten Modul Prelude, dem sog. Standard-Präludium (engl. Standard Prelude).

Das quelloffene Standard-Präludium

- wird automatisch mit jedem Haskell-Programm geladen, so dass die darin definierten Typen und Rechenvorschriften stets zur Verfügung stehen.

Tipp

Nachschlagen und lesen im Standard-Präludium ist

- gute und einfache Möglichkeit, sich mit der Syntax von Haskell vertraut zu machen und ein Gefühl für den Stil funktionaler Programmierung zu entwickeln.

'Haskell 98'-Sprachbericht:

- Simon Peyton Jones (Hrsg.). Haskell 98: Language and Libraries. The Revised Report. Cambridge University Press, 2003. `www.haskell.org/definitions`. (Kapitel 8, Standard Prelude; Kapitel 8.1, Module `Prelude`)
- Standard-Präludium. `http://www.haskell.org/onlinereport/standard-Prelude.html`

Kapitel 1.3

Nützliche Werkzeuge für Haskell: Hugs, GHC, GHCi, Hoogle, Hayoo, Leksah

Überblick

Beispielhaft 5 nützliche Werkzeuge für die funktionale Programmierung in Haskell:

1. Hugs: Ein Haskell-Interpretierer
2. GHC, GHCi: Ein Haskell-Übersetzer, ein Haskell-Interpretierer
3. Hoogle, Hayoo: Zwei Haskell(-spezifische) Suchmaschinen
4. Leksah: Eine (in Haskell geschriebene) quelloffene integrierte Entwicklungsumgebung IDE

Kapitel 1.3.1

Hugs

Hugs

...ein populärer Haskell-Interpretierer (mit allerdings eingestellter Entwicklung).

Hugs im Netz:

- `www.haskell.org/hugs`

Zur Arbeit mit Hugs siehe z.B.:

- H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. `citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf` (Chapter 4, Using the Hugs Interpreter)

Hugs-Aufruf ohne Programmskript

Aufruf von Hugs ohne Skript:

```
hugs
```

Nach Aufruf steht die Taschenrechnerfunktionalität von Hugs (sowie im Prelude definierte Funktionen) zur Auswertung von Ausdrücken zur Verfügung:

```
Main>47*100+11
4711
Main>reverse "stressed"
"desserts"
Main>length "desserts"
8
Main>(4>17) || (17+4==21)
True
Main>True && False
False
```

Hugs-Aufruf mit Programmskript

Aufruf von Hugs mit Skript:

```
hugs <filename>
```

Zum Beispiel:

```
hugs myFirstScript.hs
hugs myFirstScript.lhs
```

Nach Aufruf stehen zusätzlich zu den Prelude-Funktionen auch alle im geladenen Skript deklarierten Funktionen zur Verfügung:

```
Main>fac 6
720
Main>binom' (49,6)
13.983.816
Main>sechsAus45
8.145.060
```

Das Hugs-Kommando `:l(oad)` erlaubt ein anderes Skript zu laden (wodurch ein eventuell vorher geladenes Skript ersetzt wird):

```
Main>:l myFirstScript.lhs
```

Wichtige Hugs-Kommandos

:?	Liefert Liste der Hugs-Kommandos
:load <fileName>	Lädt die Haskell-Datei <fileName> (erkennbar an Endung .hs bzw. .lhs)
:reload	Wiederholt letztes Ladekommando
:quit	Beendet den aktuellen Hugs-Lauf
:info name	Liefert Information über das mit name bezeichnete "Objekt"
:type exp	Liefert den Typ des Argumentausdrucks exp
:edit <fileName>.hs	Öffnet die Datei <fileName>.hs enthaltende Datei im voreingestellten Editor
:find name	Öffnet die Deklaration von name im voreingestellten Editor
!<com>	Ausführen des Unix- oder DOS-Kommandos <com>

Alle Kommandos können mit dem ersten Buchstaben abgekürzt werden.

Hugs-Fehlermeldungen und -warnungen

- Fehlermeldungen
  - Syntaxfehler

    `Main> sechsAus45 == 123456)` ...liefert

    `ERROR: Syntax error in input (unexpected ‘)’)`
  - Typfehler

    `Main> sechsAus45 + False` ...liefert

    `ERROR: Bool is not an instance of class "Num"`
  - Programmfehler

    ...später
  - Modulfehler

    ...später
- Warnungen
  - Systemmeldungen

    ...später

Bibliotheken: Professionell und praxisgerecht

- Haskell stellt umfangreiche Bibliotheken mit vielen vordefinierten Funktionen zur Verfügung.
- Das sog. Standard-Präludium (engl. `Standard Prelude`) wird automatisch beim Start von Hugs geladen und stellt eine Vielzahl von Funktionen bereit, z.B. zum
  - Umkehren von Zeichenreichen bzw. genereller, allgemein von Listen beliebiger Typen (`reverse`)
  - Ver- und entpaaren von Listen (`zip`, `unzip`)
  - Aufsummieren und Aufmultiplizieren von Elementen einer numerischen Liste (`sum, product`)
  - ...

Namenskonflikte mit vordefinierten Namen

...soll eine Funktion eines gleichen (bereits in `Prelude.hs` vordefinierten) Namens deklariert werden, können Namenskonflikte durch verstecken (engl. hiding) vordefinierter Namen vermieden werden.

Am Beispiel von `reverse`, `zip`, `sum`:

- Füge die Zeile

  ```
  import Prelude hiding (reverse,zip,sum)
  ```

  am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden) ein; dadurch werden die vordefinierten Namen `reverse`, `zip` und `sum` verborgen.

  (Mehr dazu später in Kapitel 17 im Zusammenhang mit dem Modulkonzept von Haskell).

Kapitel 1.3.2

GHC, GHCi

GHC, GHCi

...ein populärer Haskell-Compiler:

- Glasgow Haskell Compiler (GHC)

...und ein damit verbundener Interpretierer:

- GHCi (GHC interactive)

GHC und GHCi im Netz:

- `hackage.haskell.org/platform`

Kapitel 1.3.3

Hoogle, Hayoo

Hoogle, Hayoo

...zwei nützliche Suchmaschinen, um vordefinierte Funktionen (in Haskell-Bibliotheken) aufzuspüren.

Hoogle und Hayoo unterstützen die Suche nach

- Funktionsnamen
- Modulnamen
- Funktionssignaturen

Hoogle und Hayoo im Netz:

- `www.haskell.org/hoogle`
- `hayoo.fh-wedel.de`

Kapitel 1.3.4

Leksah

Leksah

...eine quelloffene in Haskell geschriebene IDE mit GTK-Oberfläche für Linux, Windows und MacOS.

Unterstützte Eigenschaften:

- Quell-Editor zur Quellprogrammerstellung.
- Arbeitsbereiche zur Verwaltung von Haskell-Projekten in Form eines oder mehrerer Cabal-Projekte.
- Cabal-Paketverwaltung zur Verwaltung von Versionen, Übersetzeroptionen, Testfällen, Haskell-Erweiterungen, etc.
- Modulbetrachter zur Projektinspektion.
- Debugger auf Basis eines integrierten ghc-Interpretierers.
- Erweiterte Editorfunktionen mit Autovervollständigung, 'Spring-zu-Fehlerstelle'-Funktionalität, etc.
- ...

Leksah (fgs.)

Leksah im Netz:

- www.leksah.org

Anmerkung:

- Teils aufwändige Installation, oft vertrackte Versionsabhängigkeiten zwischen Komponenten.
- Für die Zwecke der LVA nicht benötigt.

Kapitel 1.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (1)

- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
- John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
- Henri E. Baal, Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994. (Chapter 4, Functional Languages; Chapter 7, Other Paradigms)
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 1, Motivation und Einführung)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (2)

- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 1, What is functional programming? Kapitel 2.1, A session with GHCi)

- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 1, Einführung; Kapitel 2, Programmierumgebung; Kapitel 4.1, Rekursion über Zahlen; Kapitel 6, Die Unix-Programmierumgebung)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (3)

- H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf (Chapter 1.2, Excerpts from Backus' 1977 Turing Award Address; Chapter 1.3, Programming Language Paradigms; Chapter 1.4, Reasons for Studying Functional Programming; Chapter 1.5, Objections Raised Against Functional Programming; Chapter 4, Using the Hugs Interpreter)
- Hal Daumé III. *Yet Another Haskell Tutorial*. wikibooks.org-Ausgabe, 2007. https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (4)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 1.1, The von Neumann Bottleneck; Kapitel 1.2, Von Neumann Languages)

- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.

- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC

- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 1, Erste Schritte; Anhang A, Zur Benutzung des Systems)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (5)

- Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. tryhaskell.org.
- Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture, Communications of the ACM 22(8):455-460, 1979.
- Bastiaan Heeren, Daan Leijen, Arjan van IJzendoorn. *Helium, for Learning Haskell*. In Proceedings of the ACM SIGPLAN 2003 Haskell Workshop (Haskell 2003), 62-71, 2003.
- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
- C.A.R. Hoare. *Algorithm 64: Quicksort*. Communications of the ACM 4(7):321, 1961.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (6)

- C.A.R. Hoare. *Quicksort*. The Computer Journal 5(1):10-15, 1962.
- Paul Hudak, Joseph Fasel, John Peterson. *A Gentle Introduction to Haskell*. Technischer Bericht, Yale University, 1996. `https://www.haskell/org/tutorial`
- John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.
- Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 1, Introduction; Kapitel 2, First Steps)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (7)

- Arjan van IJzendoorn, Daan Leijen, Bastiaan Heeren. *The Helium Compiler*. www.cs.uu.nl/helium.
- Mark P. Jones, Alastair Reid et al. *The Hugs98 User Manual*, 1999. `www.haskell.org/hugs`.
- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
- Donald Knuth. *Literate Programming*. The Computer Journal 27(2):97-111, 1984.
- Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (8)

- Martin Odersky. *Funktionale Programmierung*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 599-612, 2006. (Kapitel 5.1, Funktionale Programmiersprachen; Kapitel 5.2, Grundzüge des funktionalen Programmierens)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 1, Getting Started)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 1, Was die Mathematik uns bietet; Kapitel 2, Funktionen als Programmiersprache)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (9)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 1, Grundlagen der funktionalen Programmierung)
- Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*, Susan Eisenbach (Hrsg.), Ellis Horwood, 9-20, 1987.
- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (10)

- Simon Thompson. *Where Do I Begin? A Problem Solving Approach in Teaching Functional Programming*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-Verlag, LNCS 1292, 323-334, 1997.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 1, Introducing functional programming; Kapitel 2, Getting started with Haskell and Hugs)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 1, Introducing functional programming; Kapitel 2, Getting started with Haskell and GHCi)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (11)

- Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. `www.haskell.org/definitions`. (Kapitel 8, Standard Prelude; Kapitel 8.1, Module `Prelude`)
- Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3, Functional Programming Languages; Kapitel 3.1, Basic Concepts and Introductory Examples)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 1 (12)

- Hugs-Benutzerhandbuch. *The Hugs98 User Manual.* `https://www.haskell.org/hugs/pages/hugsman/index.html`
- GHCi-Benutzerhandbuch. *Glasgow Haskell Compiler User's Guide.* `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html`
- Haskells Standard-Präludium. `https://www.haskell.org/onlinereport/standard-prelude.html`

Teil II

Grundlagen

Kapitel 2

Elementare Typen, Tupel, Listen, Zeichenreihen

Kapitel 2.1

Elementare Typen

Überblick

Elementare (Daten-) Typen

- Wahrheitswerte: `Bool`
- Ganze Zahlen: `Int`, `Integer`
- Gleitkommazahlen: `Float`, `Double`
- Zeichen: `Char`

...in der Folge nach folgendem Schema angegeben:

- Typname
- Typtypische Konstanten
- Typtypische Operatoren und Relatoren

Für Details siehe Haskell-Sprachbericht und Standard-Präludium.

Kapitel 2.1.1

Wahrheitswerte

Wahrheitswerte: Bool

Typ	`Bool`	Wahrheitswerte
Konstanten	`True :: Bool` `False :: Bool`	Darstellung v. ‘wahr’ Darstellung v. ‘falsch’
Spezialwert	`otherwise :: Bool` `otherwise = True`	(Bedingte Ausdrücke: Stets erfüllter Wächter)

Operatoren	`(&&) :: Bool -> Bool -> Bool`
	`(||) :: Bool -> Bool -> Bool`
	`not :: Bool -> Bool`
		Konjunktion
		Disjunktion
		Negation

Relatoren	`(==) :: Bool -> Bool -> Bool`
	`(/=) :: Bool -> Bool -> Bool`
	`(>) :: Bool -> Bool -> Bool`
	...
		gleich
		ungleich
		echt größer

Kapitel 2.1.2

Ganze Zahlen

Ganze Zahlen: Int, Integer (1)

Typ	`Int`	Ganze Zahlen, -2^{63} bis $2^{63}-1$ oder -2^{31} bis $2^{31}-1$ (engl. fixed-precision integers)
Konstanten	`0  :: Int`	Null
	`42 :: Int`	Zweiundvierzig
	`-5 :: Int`	Minus fünf
	…	
Operatoren	`(+) :: Int -> Int -> Int`	Addition
	`(*) :: Int -> Int -> Int`	Multiplikation
	`(^) :: Int -> Int -> Int`	Exponentiation
	`(-) :: Int -> Int -> Int`	Subtraktion
	` -  :: Int -> Int`	Vorzeichenwechsel
	`div :: Int -> Int -> Int`	Division
	`mod :: Int -> Int -> Int`	Divisionsrest
	`abs :: Int -> Int`	Absolutbetrag
	`negate :: Int -> Int`	Vorzeichenwechsel
	…	
	`fromInt :: Num a => Int -> a`	Typkonversion

Ganze Zahlen: `Int`, `Integer` (2)

Relatoren

```
(==) :: Int -> Int -> Bool    gleich
(/=) :: Int -> Int -> Bool    ungleich
(>=) :: Int -> Int -> Bool    größer oder gleich
 (>) :: Int -> Int -> Bool    echt größer
(<=) :: Int -> Int -> Bool    kleiner oder gleich
 (<) :: Int -> Int -> Bool    echt kleiner
...
```

Vorgriff: Numerische Typen in Haskell: Ganze Zahlen, Gleitkommazahlen, rationale und komplexe Zahlen, zusammengefasst in der Typklasse `Num` (vgl. `fromInt :: Num a => Int -> a`)

Ganze Zahlen: `Int`, `Integer` (3)

Typ	`Integer`	Ganze Zahlen, keine Bereichsbeschränkung (engl. arbitrary-precisision integers)
Konstanten	`0 :: Integer`	Null
	`42 :: Integer`	Zweiundvierzig
	`-5 :: Integer`	Minus fünf
	`93948307853803234 :: Integer`	'Große' Zahl
	...	
Operatoren	`(+) :: Integer -> Integer -> Integer`	Addition
	...	
	`fromInteger :: Num a => Integer -> a`	Typkonversion
Relatoren	...	

Konstanten, Operatoren und Relatoren für `Integer` wie für `Int`, jedoch keine *a-priori* Zahlbereichsbeschränkung.

Kapitel 2.1.3

Gleitkommazahlen

Gleitkommazahlen: `Float`, `Double` (1)

Typ	`Float`	Gleitkommazahlen (32 Bit-Darstellung)
Konstanten	`0.125    :: Float`	Ein Achtel
	`-1.75   :: Float`	Minus eindreiviertel
	`8.5e-2 :: Float`	Achteinhalbhundertstel
	...	
Operatoren	`(+) :: Float -> Float -> Float`	Addition
	`(*) :: Float -> Float -> Float`	Multiplikation
	...	
	`sqrt :: Float -> Float`	(positive) Quadrat-wurzel
	`sin  :: Float -> Float`	sinus
	`cos  :: Float -> Float`	cosinus
	...	
	`ceiling :: Float -> Int`	Typkonversion
	`floor   :: Float -> Int`	Typkonversion
	`round   :: Float -> Int`	Typkonversion

Gleitkommazahlen: `Float`, `Double` (2)

Relatoren

```
(==) :: Float -> Float -> Bool
(/=) :: Float -> Float -> Bool
(>=) :: Float -> Float -> Bool
 (>) :: Float -> Float -> Bool
(<=) :: Float -> Float -> Bool
 (<) :: Float -> Float -> Bool
...
```

`(==)`	gleich
`(/=)`	ungleich
`(>=)`	größer oder gleich
`(>)`	echt größer
`(<=)`	kleiner oder gleich
`(<)`	echt kleiner

Gleitkommazahlen: `Float`, `Double` (3)

Typ	`Double`	Gleitkommazahlen (64 Bit-Darstellung)
Konstanten	...	
Operatoren	...	
Relatoren	...	

Konstanten, Operatoren und Relatoren für `Double` wie für `Float`, jedoch mit doppelter Genauigkeit.

Kapitel 2.1.4

Zeichen, Ziffern, Sonderzeichen

Zeichen: Char

Typ	`Char`	Zeichen (Literal) (Unicode-Darst.)
Konstanten	`'a' :: Char`	Darst. von a
	…	
	`'Z' :: Char`	Darst. von Z
	`'\t' :: Char`	Tabulator
	`'\n' :: Char`	Neue Zeile
	`'\\' :: Char`	'backslash'
	`'\'' :: Char`	Hochkomma
	`'\"' :: Char`	Anführungszeichen
	…	
Operatoren	`ord :: Char -> Int`	Konversionsfkt.
	`chr :: Int -> Char`	Konversionsfkt.
	…	
Relatoren	`(==) :: Char -> Char -> Bool`	gleich
	`(>) :: Char -> Char -> Bool`	echt größer
	…	

Kapitel 2.2

Tupel

Tupel

Tupel

- fassen eine vorbestimmte Zahl von Werten möglicherweise verschiedener Typen zusammen.
- sind in diesem Sinn heterogen.

Allgemeines Muster

- Allgemeines Muster für Tupelwerte

  `(v1,v2,...,vk) :: (T1,T2,...,Tk)`

  Dabei bezeichnen `v1,...,vk` Werte und `T1,...,Tk` Typen mit

  `v1 :: T1, v2 :: T2,..., vk :: Tk`

- Standardkonstruktor (runde Klammern)
  - Leerer Tupelkonstruktor:

    `() :: ()` (exotisch, aber sinnvoll, s. Kap. 15)
  - Paarkonstruktor:

    `(,) :: a -> b -> (a,b)`
  - Tripelkonstruktor:

    `(,,) :: a -> b -> c -> (a,b,c)`
  - Quadrupelkonstruktor:

    `(,,,) :: a -> b -> c -> d -> (a,b,c,d)`
  - ...

Beispiele für Tupel

Beispiele:

```
(,) 3.14 17.4 ->> (3.14,17.4) :: (Float,Float)
(,) 'a' True  ->> ('a',True) :: (Char,Bool)
(,) "Fun" 3   ->> ("Fun",3) :: (String,Int)

(,) ("Fun",3) True
     ->> (("Fun",3),True) :: ((String,Int),Bool)

(,,) 5 8 6.5 ->> (5,8,6.5) :: (Int,Int,Float)

(,,,) 'b' False "Fun" 3
     ->> ('b',False,"Fun",3) :: (Char,Bool,String,Int)

p = (,,,) 'b' False :: a -> b -> (Char,Bool,a,b)
p "Fun" 3
     ->> ('b',False,"Fun",3) :: (Char,Bool,String,Int)
p 2.1 True
     ->> ('b',False,2.1,True) :: (Char,Bool,Float,Bool)
```

Standardselektoren für Paare

Standardselektoren (vordefiniert ausschließlich für Paare):

```
fst :: (a,b) -> a
fst (x,_) = x
snd :: (a,b) -> b
snd (_,y) = y
```

Aufrufbeispiele:

```
fst (3.14,'a') ->> 3.14
snd (3.14,'a') ->> 'a'
```

Bemerkung: Für drei- und mehrstellige Tupel bietet Haskell keine vordefinierten (Selektor-) Funktionen für den Zugriff auf Tupelkomponenten an.

Selektoren für Tripel

Selbstdefinierte Selektoren für Tripel:

```
fst' :: (a,b,c) -> a
fst' (x,_,_) = x
snd' :: (a,b,c) -> b
snd' (_,y,_) = y
thd' :: (a,b,c) -> c
thd' (_,_,z) = z
```

Aufrufbeispiele:

```
fst' (3.14,'a',True) ->> 3.14
snd' (3.14,'a',True) ->> 'a'
thd' (3.14,'a',True) ->> True
```

Kapitel 2.3

Listen

Listen

Listen

- fassen eine nicht vorbestimmte Zahl von Werten gleichen Typs zusammen.
- sind in diesem Sinn homogen.

Allgemeines Muster

- Allgemeines Muster für Listenwerte

  `v1 : (v2 : (.. : (vk : [])..)) = [v1,v2,..,vk] :: [T]`

  Dabei bezeichnen `v1,...,vk` Werte und `T` einen Typ mit `v1, v2,..., vk :: T`

- Standardkonstruktor (:), zusätzlich eckige Klammern als Listenoperator für kompaktere Schreibweise
  - Allgemein:

    `v1 : (v2 : (.. : (vk : [])..)) :: [a]` (Standard)

    `[v1,v2,..,vk] :: [a]` (Abgekürzt)
  - Konkret:

    `1 : (2 : (3 : (4 : []))) :: [Int]` (Standard)

    `[1,2,3,4] :: [Int]` (Abgekürzt)

    `[] :: [a]` (Leere Liste)

  Erinnerung:
  ```
  quickSort []     = []
  quickSort (n:ns) = ...
  ```

Beispiele für Listen (1)

...von

- Zeichen
  `['a','b','c','d','e'] :: [Char]`
- Wahrheitswerten
  `[True,False,True] :: [Bool]`
- ganzen Zahlen
  `[2,5,17,4,42,4711] :: [Int]`
- Gleitkommazahlen
  `[3.14,5.0,12.21] :: [Float]`
- keinen Elementen, leere Liste
  `[]`

Beispiele für Listen (2)

...von

- Tupeln

  ```
  [('a',True),('b',False),('c',False),
   ('d',False),('e',True)] :: [(Char,Bool)]
  ```

  ```
  [(3,5,4.0),(4,7,5.5),(2,8,5.0),(2,11,6.5)]
                                :: [(Int,Int,Float)]
  ```

- Listen

  ```
  [[1,2,3],[42],[],[17,4,21],[],[3,2,1]] :: [[Int]]
  ```

  ```
  [(['f','p'],2),(['h'],1),([],0)] :: [([Char],Int)]
  ```

  ```
  [("fun",3),("h",1),("",0)] :: [([Char],Int)]
  ```

Beispiele für Listen (3)

...von

- Zeichenreihen

  `["sin","cos","tan","sin","cos","tan"] :: [[Char]]`

- Funktionen

  `[sin,cos,tan,sin,cos,tan] :: [Float -> Float]`

  `[(+),(*),ggt,mod] :: [Int -> Int -> Int]`

  `[binom,binom] :: [Integer -> Integer -> Integer]`

  `[binom'] :: [(Integer,Integer) -> Integer]`

- ...

Vergleichbarkeit und Gleichheit von Listen (1)

Nur typgleiche Listenwerte sind vergleichbar:

```
cs = ['a','b','c'] :: [Char]
ss = ["a","b","c"] :: [Strings]
xs = "abc" :: String

ns = [1,2,3] :: [Int]
ms = [1,2,3] :: [Integer]

fs = [1.0,2.0,3.0] :: [Float]
ds = [1.0,2.0,3.0] :: [Double]

cs == ns ->> "Fehler: Typfehler in Anwendung"
ns == fs ->> "Fehler: Typfehler in Anwendung"
ns == ms ->> "Fehler: Typfehler in Anwendung"
fs == ds ->> "Fehler: Typfehler in Anwendung"

ns == ns ->> True
fs == fs ->> True

cs == ss ->> "Fehler: Typfehler in Anwendung"
ss == xs ->> "Fehler: Typfehler in Anwendung"

cs == xs ->> True
```

Vergleichbarkeit und Gleichheit von Listen (2)

Nur typgleiche Listen gleicher Länge u. Anordnung sind gleich:

```
ns = [1,2,3,4]
ms = [1,2,3]
ks = [1,2,3,4,5]
ls = [2,1,4,3]

ns == ns ->> True

ns == ms ->> False
ns == ks ->> False
ns == ls ->> False
```

Vergleichbarkeit und Gleichheit von Listen (3)

Auch “leere” Listen sind nur typabhängig vergleichbar:

```
[] == [] ->> True
[] :: [Int] == [] :: [Int] ->> True
[] :: [Int] == [] :: [Integer]
    ->> "Fehler: Typfehler in Anwendung"
[] :: [Float] == [] :: [Double]
    ->> "Fehler: Typfehler in Anwendung"
```

```
bs = [] :: [Bool]
cs = [] :: [Char]
bs == cs ->> "Fehler: Typfehler in Anwendung"
bs /= cs ->> "Fehler: Typfehler in Anwendung"
```

```
xs = []
ys = []
xs == ys ->> True
xs /= ys ->> False
```

Vordefinierte Funktionen auf Listen (1)

Name und Typ	Bedeutung und Beispiel
`(:) :: a -> [a] -> [a]`	Anfügen eines Elements am Anfang einer Liste: `5:[3,2] ->> [5,3,2]`
`(++) :: [a] -> [a] -> [a]`	Aneinanderhängen zweier Listen: `[11,7] ++ [5,3,2] ->> [11,7,5,3,2]`
`(!!) :: [a] -> Int -> a`	Zugreifen auf ein Listenelement: `[5,3,2]!!0 ->> 5` `[5,3,2]!!1 ->> 3`
`concat :: [[a]] -> [a]`	Verflachen einer Liste von Listen zu einer Liste: `concat [[11,7],[5,3,2]] ->> [11,7,5,3,2]`
`reverse :: [a] -> [a]`	Umkehren einer Liste: `reverse [5,3,2] ->> [2,3,5]`

...und viele mehr (siehe Standard-Präludium).

Vordefinierte Funktionen auf Listen (2)

...drei Beispiele vordefinierter Funktionen auf Listen mit ihrer Implementierung.

Die Funktion `length`:

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Aufrufbeispiele:

```
length [1,2,3]                 ->> 3
length [[1],[2,3],[4,5,6]]     ->> 3
length [sin,cos,tan]           ->> 3
length []                      ->> 0
```

Vordefinierte Funktionen auf Listen (3)

Die Funktionen `head` und `tail`:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Aufrufbeispiele:

```
head [1,2,3]                 ->> 1
head [[1],[2,3],[4,5,6]]     ->> [1]
head [sin,cos,tan]          "->> sin"
head [sin,cos,tan] (pi/2)    ->> 1.0

tail [1,2,3]                 ->> [2,3]
tail [[1],[2,3],[4,5,6]]     ->> [[2,3],[4,5,6]]
tail [sin,cos,tan]          "->> [cos,tan]"
```

Listenkomprehension

...Zusammenfassung, Vereinigung von Mannigfaltigkeiten zu einer Einheit (Philos.):

- In funktionalen Sprachen ein wichtiges und ausdrucks-kräftiges Sprachkonstrukt, das eine automatische Generierung von Listen unterstützt!

Beispiele:

```
ns = [1..10]  -- kurz für [1,2,3,4,5,6,7,8,9,10]

[3*n | n <- ns] ->> [3,6,9,12,15,18,21,24,27,30]
[n | n <- ns, odd(n)] ->> [1,3,5,7,9]
[n | n <- ns, even(n), n>5] ->> [6,8,10]
[n*(n+1) | n <- ns, (even(n) || n>5)]
                    ->> [6,20,42,56,72,90,110]
[p | n <- ns, m <- ns, n<=3, m>=9, let p=m*n]
                    ->> [9,10,18,20,27,30]
```

Aufzählungsausdrücke

...stellen einen weiteren Erzeugungsautomatismus für Listen über Typen geordneter aufzählbarer Werte (Typen der Typklasse `Enum`) dar:

```
[2..13]       ->> [2,3,4,5,6,7,8,9,10,11,12,13]
[2,5..20]     ->> [2,5,8,11,14,17,20]
[2,5..21]     ->> [2,5,8,11,14,17,20]
[2,5..22]     ->> [2,5,8,11,14,17,20]
[2,5..23]     ->> [2,5,8,11,14,17,20,23]
[11,9..3]     ->> [11,9,7,5,3]
[11,9..2]     ->> [11,9,7,5,3]
[11,10,..2]   ->> [11,10,9,8,7,6,5,4,3,2]
[11..2]       ->> []
['a','c'..'g'] ->> ['a','c','e','g'] ->> "aceg"
['a','c'..'h'] ->> ['a','c','e','g'] ->> "aceg"
[0.0,0.3..1.2] ->> [0.0,0.3,0.6,0.9,1.2]
```

Beachte

...folgende Gleichheiten und abkürzende Schreibweisen:

```
   (1:(2:(3:[])))            (Standarddarstellung)
== 1:2:3:[]                 (Rechtsassoziativität)
== [1,2,3]                  (Syntaktischer Zucker)
```

...Typisierungen und überladene Schreibweisen:

```
[] :: [a]

[1,2,3]
ns = [1,2,3] :: [Int]
ms = [1,2,3] :: [Integer]

[1,2,3] :: Num a => [a]     (Gleiche Schreibweise,
ns :: [Int]                    verschiedene Typen,
ms :: [Integer]             Überladung von [1,2,3])
```

...überprüfbar in Hugs mittels des Kommandos :t.

Kapitel 2.4

Zeichenreihen

Zeichenreihen

Zeichenreihen

- fassen eine nicht vorbestimmte Zahl von Werten des Typs Zeichen `Char` zusammen.
- sind spezielle Listen und deshalb ebenfalls homogen.

Zeichenreihen (engl. Strings)

Zeichenreihen sind in Haskell über dem Datentyp Liste realisiert, als Listen von Zeichen, kurz Zeichenlisten:

Typ	`[Char]`	Zeichenlisten
Bezeichner	`String`	Typsynonym

Vereinbarung	`"data [a] = [] | a:[a]`
	`deriving (Eq,Ord)"`
	`type String = [Char]`
		Kein zulässiges Haskell; nur zur Illustration

Konstanten	`['F','u','n'] :: String`
	`"Fun" :: String`
	`[] :: String`
	`"" :: String`
		Zwei Darst. der Z-Reihe 'Fun' und der leeren Z-Reihe

Operatoren	`(++) :: String -> String -> String` ...	Konkatenation
Relatoren	`(==) :: String -> String -> Bool` `(/=) :: String -> String -> Bool`	gleich ungleich

Beispiele für Zeichenreihen

Beispiele:

```
['h','e','l','l','o'] == "hello" ->> True
['h','e','l','l','o'] ++ ", world"
                  == "hello, world" ->> True

"a" == 'a'   ->> "Fehler: Typfehler in Anwendung"
"a" == ['a'] ->> True
"a" == []    ->> False
```

Vordefinierte Funktionen auf Zeichenreihen

...Zeichenreihen sind Listen über dem Zeichentyp `Char`, mithin Listen:

```
type String = [Char]
```

Deshalb stehen alle auf Listen vordefinierte Operatoren und Relatoren unmittelbar auch auf Zeichenreihen zur Verfügung.

Kapitel 2.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 2 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 2, Einfache Datentypen; Kapitel 5.1, Listen; Kapitel 5.2, Tupel; Kapitel 5.3, Zeichenreihen)
- Richard Bird. *Introduction to Functional Programming using Haskell*. Cambridge University Press, 2. Auflage, 1998. (Kapitel 2, Simple datatypes; Kapitel 4, Lists)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types, and values; Kapitel 4, Lists)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 1, Elemente funktionaler Programmierung)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 2 (2)

- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.1, Basic concepts; Kapitel 3.2, Basic types; Kapitel 3.3, List types; Kapitel 3.4, Tuple types; Kapitel 5, List comprehensions)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 1, An Intro to Lists, Tuples; Kapitel 2, Common Haskell Types)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions – Useful Composite Data Types: Lists and Tuples, Functions over Lists and Tuples)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 2 (3)

- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 3.2, Elementare Strukturen; Kapitel 15, Listen (Sequenzen))
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 3, Basic types and definitions; Kapitel 5, Data types, tuples and lists)

Kapitel 3

Funktionen

Kapitel 3.1

Definition, Schreibweisen, Sprachkonstrukte

Funktionen

...sind wichtigstes Abstraktions- und Ausdrucksmittel in funktionaler Programmierung.

Funktionale Programmiersprachen bieten deshalb oft mehrere Schreibweisen an, um Funktionen zu definieren. So ist

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

nur eine Möglichkeit, die Fakultätsfunktion in Haskell zu definieren:

$$! : \mathbb{N} \rightarrow \mathbb{N}$$

$$n\,! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{sonst} \end{cases}$$

Prägnanz d. Vermeiden bedingter Ausdrücke

Haskell bietet weitere Schreibweisen an, die meist knapper, konziser und deshalb übersichtlicher und verständlicher sind, insbesondere durch weitere Möglichkeiten

- Fallunterscheidungen

anders als durch bedingte Ausdrücke wie in

$$\texttt{fac :: Int -> Int}$$

$$\texttt{fac n = }\underbrace{\texttt{if n == 0 then 1 else n * fac (n-1)}}_{\textit{Bedingter Ausdruck}}$$

auszudrücken, insbesondere

- bewachte Ausdrücke
- Muster

Schreibweisen für Funktionsdefinitionen (1)

(1) Mittels bedingter Ausdrücke:

```
fac :: Int -> Int
```

$$\texttt{fac n = }\underbrace{\texttt{if n == 0 then 1 else n * fac (n-1)}}_{\textit{Bedingter Ausdruck}}$$

(2) Mittels bewachter Ausdrücke:

```
fac :: Int -> Int
fac n
```

$$\texttt{|}\ \underbrace{\texttt{n == 0}}_{\textit{Wächter}}\ \underbrace{\texttt{= 1}}_{\textit{Bewachter Ausdruck}}$$

$$\texttt{|}\ \underbrace{\texttt{otherwise}}_{\textit{Wächter}}\ \underbrace{\texttt{= n * fac (n-1)}}_{\textit{Bewachter Ausdruck}}$$

(otherwise, stets erfüllter Wächter)

```
otherwise :: Bool
otherwise = True
```

Schreibweisen für Funktionsdefinitionen (2)

(3a) Mittels Muster (hier für ganze Zahlen):

```
fac :: Int -> Int              --Fakultätsfunktion
fac 0 = 1
fac n = n * fac (n - 1)

fib :: Int -> Int              --Fibonacci-Funktion
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Schreibweisen für Funktionsdefinitionen (3)

(3b) Mittels Muster (hier für Zeichen):

```
capitalizeVowels :: Char -> Char
capitalizeVowels = capVow

capVow :: Char -> Char
capVow 'a' = 'A'
capVow 'e' = 'E'
capVow 'i' = 'I'
capVow 'o' = 'O'
capVow 'u' = 'U'
capVow  c  =  c
```

(3c) Mittels Muster (hier für Wahrheitswerte):

```
xor :: Bool -> Bool -> Bool
xor True  False = True
xor False True  = True
xor b1    b2    = False
```

Schreibweisen für Funktionsdefinitionen (4)

(3d) Mittels Muster (hier für Listen):

```
quickSort :: [Integer] -> [Integer]
quickSort []      = []
```

Muster leere Liste

```
quickSort (n       :       ns)
```

Muster Listenkopf *Muster Listenrest*

Muster nichtleere Liste

```
                  = quickSort [m | m <- ns, m <= n]
                    ++ [n]
                    ++ quickSort [m | m <- ns, m > n]
```

Schreibweisen für Funktionsdefinitionen (5)

(3e) Mittels Muster, hier zusätzlich mit “wildcard”-Muster:

```
mult :: Int -> Int -> Int
mult 0 _ = 0
mult _ 0 = 0
mult 1 y = y
mult x 1 = x
mult x y = x*y

tail :: [a] -> [a]
tail (_:xs) = xs
tail _      = error "Liste darf nicht leer sein."

nand :: Bool -> Bool -> Bool
nand False _ = True
nand _ False = True
nand _ _     = False    (auch in xor ist _ mögl.)
```

Schreibweisen für Funktionsdefinitionen (6)

Mittels (Muster und...)

(4) case-Ausdrucks:

```
capVow :: Char -> Char
capVow c = case c of 'a' -> 'A'
                     'e' -> 'E'
                     'i' -> 'I'
                     'o' -> 'O'
                     'u' -> 'U'
                     otherwise -> c

describeList :: [a] -> String
describeList ls
 = "The list ls "
    ++ case ls of []         -> "is empty."
                  (x:[])     -> "is a singleton list."
                  (x:y:[])   -> "has two elements."
                  _ -> "has three or more elements."
```

Schreibweisen für Funktionsdefinitionen (7)

Mittels (Muster und...)

(5a) lokaler Deklarationen (where-Konstrukt, nachgestellt):

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) = quickSort smaller
                   ++ [n]
                   ++ quickSort larger
                   where
                    smaller = [m | m<-ns, m<=n]
                    larger  = [m | m<-ns, m>n]
```

Schreibweisen für Funktionsdefinitionen (8)

Mittels (Muster und...)

(5b) lokaler Deklarationen (`let`-Konstrukt, vorgestellt):

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) = let
                    smaller = [m | m<-ns, m<=n]
                    larger  = [m | m<-ns, m>n]
                   in (quickSort smaller
                       ++ [n]
                       ++ quickSort larger)
```

Schreibweisen für Funktionsdefinitionen (9)

In einer Zeile mittels (where-Konstrukts und...)

(6a) Semikolons ";":

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) =
 quickSort smaller ++ [n] ++ quickSort larger
 where smaller = [m|m<-ns, m<=n]; larger = [m|m<-ns, m>n]
```

In einer Zeile mittels (let-Konstrukts und...)

(6b) Semikolons ";":

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) =
 let smaller = [m|m<-ns, m<=n]; larger = [m|m<-ns, m>n]
 in (quickSort smaller ++ [n] ++ quickSort larger)
```

Schreibweisen für Funktionsdefinitionen (10)

(7a) Mittels anderer Funktionen (argumentbehaftet):

```
fac :: Int -> Int
fac n = foldl (*) 1 [1..n]

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs


fac :: Int -> Int
fac n = product [1..n]

product :: (Num a) => [a] -> a
product = foldl (*) 1
```

Schreibweisen für Funktionsdefinitionen (11)

(7b) Mittels anderer Funktionen (argumentlos):

```
factorial :: Int -> Int
factorial = fac

qs :: [Integer] -> [Integer]
qs = quickSort
```

...wenn z.B. der Name `fac` zu wenig sprechend, der Name `quickSort` zu lang erscheint (vgl. auch das Funktionenpaar `capitalizeVowels` und `capVow`).

Schreibweisen für Funktionsdefinitionen (12)

(7b) Mittels anderer Funktionen (argumentlos), fgs.:

```
and :: Bool -> Bool -> Bool
and = (&&)
```

Zusätzlich zu Ausdrücken der Form

`(x>0) && (y<0)` und `(&&) (x>0) (y<0)`

sind nun auch Ausdrücke der Form

`(x>0) 'and' (y<0)` und `and (x>0) (y<0)` möglich!

```
(./.) :: Integer -> Integer -> Integer
(./.) = div
```

Zusätzlich zu Ausdrücken der Form

`div 5 2` und `5 'div' 2`

sind nun auch Ausdrücke der Form

`5 ./. 2` und `(./.) 5 2` möglich!

Schreibweisen für Funktionsdefinitionen (13)

(8) Mittels anonymer Funktionen (argumentlos):

$$\texttt{fac = } \underbrace{\texttt{\textbackslash n -> (if n == 0 then 1 else n * fac (n-1))}}_{\textit{Anonyme } \lambda\textit{-Abstraktion}}$$

Die Schreibweise ist

- Reminiszenz an den funktionaler Programmierung zugrundeliegenden λ-Kalkül:
  - Im λ-Kalkül: $\lambda x\, y.\, (x + y)$
  - In Haskell: `\x y -> x+y`

Anwendung in Haskell:

- Immer dann, wenn der Funktionsname keine Rolle spielt:

```
map (\n -> 2*n+1) [1,2,3] ->> [3,5,7]
map (\n -> n*n-1) [1,2,3] ->> [0,3,8]
```

Verwendungshinweise: Bewachte Ausdrücke (1)

Funktionen sind außer in den einfachsten Fällen fast immer über Fallunterscheidungen definiert.

- Bewachte Ausdrücke führen meist zu besserer Lesbarkeit als (geschachtelte) bedingte Ausdrücke.

Vergleiche

```
signum :: Int -> Int
signum n
 | n < 0  = -1
 | n == 0 = 0
 | n > 0  = 1
```

```
signum :: Int -> Int
signum n
 | n < 0     = -1
 | n == 0    = 0
 | otherwise = 1
```

(Ein Tick effizienter!)

mit

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Verwendungshinweise: Bewachte Ausdrücke (2)

Mischformen möglich, aber mindestens auf die Weise wie hier gar nicht sinnvoll:

```
signum :: Int -> Int
signum n
 | n < 0     = -1
 | otherwise = if n == 0 then 0 else 1
```

Verwendungshinweise: Muster (1)

Funktionen arbeiten häufig auf strukturierten Werten.

- Musterbasierte Definitionen sind meist am zweckmäßigsten und übersichtlichsten.

Vergleiche

```
binom' :: (Int,Int) -> Int
binom' (n,k)
 | k==0 || n==k = 1
 | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

mit

```
binom' :: (Int,Int) -> Int
binom' p
 | snd(p) == 0 || fst(p)==snd(p) = 1
 | otherwise = binom' (fst(p)-1,snd(p)-1)
                 + binom' (fst(p)-1,snd(p))
```

Verwendungshinweise: Muster (2)

Vorteile musterbasierter Funktionsdefinitionen:

Muster

- legen die Struktur des (Argument-) Werts offen.
- legen Namen für die verschiedenen Strukturteile des Werts fest und erlauben über diese Namen unmittelbaren Zugriff auf diese Teile.
- vermeiden dadurch sonst nötige Selektorfunktionen.

und führen so zu einem Gewinn an Lesbarkeit und Transparenz.

Bezeichnungskonventionen für Muster (1)

Für Int(eger)-Werte: n, m,...
für Listen von Int(eger)-Werten: ns, ms,...

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) = quickSort [m | m <- ns, m <= n]
                   ++ [n]
                   ++ quickSort [m | m <- ns, m > n]
```

Für Werte beliebigen Typs: x, y,...
für Listenwerte beliebigen Typs: xs, ys,...

```
quickSort :: Ord a => [a] -> [a]
quickSort []     = []
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                   ++ [x]
                   ++ quickSort [y | y <- xs, y > x]
```

Bezeichnungskonventionen für Muster (2)

- Für Zeichen-Werte: `c`, `c'`, `d`,...
  für Listen von Zeichen-Werten: `cs`, `ds`,...
- Für Ziffern-Werte: `d`, `d'`, `e`,...
  für Listen von Ziffern-Werten: `ds`, `es`,...
- Für Wahrheitswerte: `b`, `b'`,...
  für Listen von Wahrheitswerten: `bs`,...
- Für ganze Zahlen: `n`, `m`,...
  für Listen ganzer Zahlen: `ns`, `ms`,...
- Für Werte beliebigen Typs: `x`, `x'`, `y`, `y'`,...
  für Listen von Werten beliebigen Typs: `xs`, `ys`,...
- Für Listen von Listen-Werten: `nss`, `mss`, `xss`, `yss`,...
- ...

Muster

...sind (u.a. und soweit wie jetzt eingeführt):

- Konstanten eines Typs (z.B. `0`, `42`, `3.14`, `False`, `True`, `'c'`, `"c"`, `"fun"`, `""`, `[]`,...)
  ...ein Argumentwert passt mit dem Muster zusammen, wenn es wertgleich mit der Konstanten ist.
- Variablen (z.B. `n`, `x`, `c`,...)
  ...jeder Argumentwert passt.
- Wild card "_"
  ...jeder Argumentwert passt (Verwendung von "_" für alle Argumentwerte, die nicht zum Ergebnis beitragen; siehe `mult`, `tail`, `nand`,...).
- Zusammengesetzte Muster, bis jetzt für Tupel und Listen (z.B. `[]`, `[x]`, `(x:[])`, `(x:y:[])`, `(x:y:z:[])`, `(x:xs)`, `(x:y:xs)`, `(m,n)`, `(m,_)`, `(_,_)`, `(x,y,z)`, `(x,_,z)`,...)
- ...

Kapitel 3.2

Funktionssignaturen, Funktionsterme, Funktionsstelligkeiten

Überblick

Funktions-

- Signaturen
- Terme
- Stelligkeiten

und damit verbundene

- Klammereinsparungsregeln in Haskell.

Das Wichtigste auf einen Blick:

- (Funktions-) Signaturen sind rechtsassoziativ geklammert
- (Funktions-) Terme sind linksassoziativ geklammert
- (Funktions-) Stelligkeit ist 1

Beispiel: Die Editorfunktion “ersetze”

...eine Funktion, die in einem Text das n-te Vorkommen einer Zeichenreihe s durch eine Zeichenreihe s' ersetzt.

Implementierung in Haskell

```
type Txt  = String
type Vork = Int
type Alt  = Txt
type Neu  = Txt

ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

...angewendet auf einen Text t, eine Vorkommensnummer n und zwei Zeichenreihen s und s' ist das Resultat der Anwendung von ersetze ein Text, in dem das n-te Vorkommen von s in t durch s' ersetzt ist.

Eine Anwendung von “ersetze”

Funktion

```
type Txt  = String
type Vork = Int
type Alt  = Txt
type Neu  = Txt
ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

Argumente

```
"Ein rotes Rad" :: Txt
1 :: Vork
"rotes" :: Alt
"blaues" :: Neu
```

Auswertung

```
((((ersetze "Ein rotes Rad") 1) "rotes") "blaues")
 ->> "Ein blaues Rad" :: Txt
```

Schrittweise Auswertung (1)

...ersetze und die nach fortgesetzter Argumentkonsumation entstehenden Funktionsterme sind mit Ausnahme des letzten von funktionalem Typ.

Schritt 1: ersetze konsumiert **ein** Argument, den Wert "Ein rotes Rad" vom Typ Txt. Der dadurch entstehende Funktionsterm ist von funktionalem Typ:

```
(ersetze "Ein rotes Rad") ::
                        (Vork -> (Alt -> (Neu -> Txt)))
```

Schritt 2: (ersetze "Ein rotes Rad") konsumiert **ein** Argument, den Wert 1 vom Typ Vork. Der dadurch entstehende Funktionsterm ist von funktionalem Typ:

```
((ersetze "Ein rotes Rad") 1) :: (Alt -> (Neu -> Txt))
```

Schrittweise Auswertung (2)

Schritt 3: `((ersetze "Ein rotes Rad") 1)` konsumiert **ein** Argument, den Wert `"rotes"` vom Typ `Alt`. Der dadurch entstehende Funktionsterm ist von funktionalem Typ:

```
(((ersetze "Ein rotes Rad") 1) "rotes") :: (Neu -> Txt)
```

Schritt 4: `(((replace "Ein rotes Rad") 1) "rotes")` konsumiert **ein** Argument, den Wert `"blaues"` vom Typ `Neu`. Der dadurch entstehende Term ist von nichtfunktionalem Typ:

```
((((ersetze "Ein rotes Rad") 1) "rotes") "blaues") :: Txt
```

Insgesamt erhalten wir:

```
((((ersetze "Ein rotes Rad") 1) "rotes") "blaues")
  ->> "Ein blaues Rad" :: Txt
```

Funktionssignaturen, Funktionsterme

Funktionssignaturen (oder syntaktische Funktionssignaturen oder Signaturen)

- geben den Typ einer Funktion an.

Funktionsterme

- sind aus Funktionsaufrufen aufgebaute Ausdrücke.

Beispiele:

- Funktionssignatur

```
ersetze :: Txt -> Vork -> Alt -> Neu -> Txt
```

- Funktionsterme

```
ersetze "Ein rotes Rad"
ersetze "Ein rotes Rad" 1
ersetze "Ein rotes Rad" 1 "rotes"
ersetze "Ein rotes Rad" 1 "rotes" "blaues"
```

Klammereinsparungsregeln

...für Funktionssignaturen und Funktionsterme.

Rechtsassoziativität für Funktionssignaturen:

```
ersetze :: Txt -> Vork -> Alt -> Neu -> Txt
```

...steht abkürzend für die vollständig, aber nicht überflüssig rechtsassoziativ geklammerte Funktionssignatur:

```
ersetze :: (Txt -> (Vork -> (Alt -> (Neu -> Txt))))
```

Linksassoziativität für Funktionsterme:

```
ersetze "Ein rotes Rad" 1 "rotes" "blaues"
```

...steht abkürzend für den vollständig, aber nicht überflüssig linksassoziativ geklammerten Funktionsterm:

```
((((ersetze "Ein rotes Rad") 1) "rotes") "blaues")
```

Hintergrund

Die Festlegung von

- Rechtsassoziativität für Funktionssignaturen
- Linksassoziativität für Funktionsterme

dient der Einsparung von Klammern (vgl. Punkt- vor Strichrechnung in der Mathematik).

Die Festlegung erfolgt auf diese Weise, da so in

- Signaturen und Funktionstermen

meist möglichst wenige, oft gar keine Klammern nötig sind.

Stelligkeit von Funktionen in Haskell

Das Beispiel illustriert, dass Haskell-Funktionen einstellig sind:

- Es wird stets **ein** Argument zur Zeit konsumiert.

```
ersetze :: Txt -> Vork -> Alt -> Neu -> Txt

ersetze "Ein rotes Rad" :: Vork -> Alt -> Neu -> Txt
ersetze "Ein rotes Rad" 1 :: Alt -> Neu -> Txt
ersetze "Ein rotes Rad" 1 "rotes" :: Neu -> Txt
ersetze "Ein rotes Rad" 1 "rotes" "blaues" :: Txt

"Ein blaues Rad" :: Txt
```

Zur Einstelligkeit von Funktionen

Es gilt: Konsumierte Argumente müssen nicht elementar sein; ausgedrückt durch Klammerung können sie

- zusammengesetzt und komplex sein.

Beispiel:

$$\texttt{add' :: } \underbrace{\texttt{(Int -> Int)}}_{\texttt{`Arg. 1'}} \texttt{ -> } \underbrace{\texttt{(Int -> Int)}}_{\texttt{`Arg. 2'}} \texttt{ -> } \underbrace{\texttt{(Int,Int)}}_{\texttt{`Arg. 3'}} \texttt{ -> } \underbrace{\texttt{Int}}_{\texttt{`Resultat'}}$$

```
add' f g (m,n) = (+) (f m) (g n)
```

Vollständig, aber nicht überflüssig geklammert:

```
add' :: ((Int -> Int) -> ((Int -> Int) -> ((Int,Int) -> Int)))
add' f g (m,n) = (((+) (f m)) (g n))
```

Beispiel 1: Konsumation komplexer Arg. (1)

```
add' :: ((Int -> Int) -> ((Int -> Int) -> ((Int,Int) -> Int)))
add' f g (m,n) = (((+) (f m)) (g n))
```

$$\underbrace{\texttt{(add' fac fib (5,7))}}_{\texttt{:: Int}}\ \texttt{->>}$$

$$\underbrace{\underbrace{\underbrace{\texttt{(((add' fac)}}_{\texttt{:: ((Int -> Int) -> ((Int,Int) -> Int))}}\ \texttt{fib)}}_{\texttt{:: ((Int,Int) -> Int)}}\ \texttt{(5,7))}}_{\texttt{:: Int}}\ \texttt{->>}$$

$$\texttt{->>}\ \underbrace{\underbrace{\underbrace{\texttt{(((+)}}_{\texttt{:: Int -> Int -> Int}}\ \underbrace{\underbrace{\texttt{(fac}}_{\texttt{:: Int -> Int}}\ \underbrace{\texttt{5)}}_{\texttt{:: Int}}}_{\texttt{:: Int}}\texttt{)}}_{\texttt{:: Int -> Int}}\ \underbrace{\underbrace{\texttt{(fib}}_{\texttt{:: Int -> Int}}\ \underbrace{\texttt{7)}}_{\texttt{:: Int}}}_{\texttt{:: Int}}\texttt{)}}_{\texttt{:: Int}}$$

Beispiel 1: Konsumation komplexer Arg. (2)

$$\texttt{->>}\ \underbrace{(\underbrace{(\underbrace{\texttt{(+)}}_{\texttt{:: Int -> Int -> Int}}\ \underbrace{\texttt{120}}_{\texttt{:: Int}})}_{\texttt{:: Int -> Int}}\ \underbrace{\texttt{8}}_{\texttt{:: Int}})}_{\texttt{:: Int}}$$

$$\texttt{->>}\ \underbrace{(\underbrace{\texttt{(120+)}}_{\texttt{:: Int -> Int}}\ \underbrace{\texttt{8}}_{\texttt{:: Int}})}_{\texttt{:: Int}}$$

$$\texttt{->>}\ \underbrace{\texttt{128}}_{\texttt{:: Int}}$$

Beispiel 2: Konsumation komplexer Argumente

```
type I = Int

op :: (I -> I) -> I -> (I -> I) -> I -> (I -> I -> I) -> I
op f m g n h = h (f n) (g m)
```

Vollständig, aber nicht überflüssig geklammert:

```
op :: ((I -> I) -> (I -> ((I -> I) -> (I -> ((I -> (I -> I)) -> I)))))
op f m g n h = ((h (f m)) (g n))
```

Aufrufbeispiel:

```
op fac 5 fib 7 ggt ->>
                   (((((op fac) 5) fib) 7) ggt)
:: (I -> ((I -> I) -> (I -> ((I -> (I -> I)) -> I))))
    :: ((I -> I) -> (I -> ((I -> (I -> I)) -> I)))
           :: (I -> ((I -> (I -> I)) -> I))
               :: ((I -> (I -> I)) -> I)
                              :: I
 ->> ggt (fac 7) (fib 5) ->> ((ggt (fac 7)) (fib 5))
 ->> ((ggt 5040) 3)      ->> 3
```

Klammereinsparungen für Funktionsterme (1)

...anhand einiger Beispiele:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = (((fib (n-2)) + (fib (n-1))))
```

Der vollständig, aber nicht überflüssig geklammerte Ausdruck

- `(((fib (n-2)) + (fib (n-1))))`

kann bedeutungsgleich verkürzt werden zu:

- `fib (n-2) + fib (n-1)`

...aber nicht weiter:

- `fib n-2 + fib n-1` entspricht vollständig geklammert `(((fib n) - 2) + ((fib n) - 1))`
- `fib (7-2) + fib (7-1) ->> fib 5 + fib 6 ->> 3 + 5 ->> 8`
- `fib 7-2 + fib 7-1 ->> (8 - 2) + (8 - 1) ->> 13`

Klammereinsparungen für Funktionsterme (2)

Die vollständig, aber nicht überflüssig geklammerten Ausdrücke

- `((fac (fib 6)) - 1) ->> 119`
- `(fac ((fib 6) - 1)) ->> 24`
- `(fac (fib (6 - 1))) ->> 6`

können bedeutungsgleich verkürzt werden zu:

- `fac (fib 6) - 1 ->> fac 5 - 1 ->> 120 - 1 ->> 119`
- `fac (fib 6 - 1) ->> fac (5 - 1) ->> fac 4 ->> 24`
- `fac (fib (6 - 1)) ->> fac (fib 5) ->> fac 3 ->> 6`

Weitere Klammern können ohne Bedeutungsänderung nicht eingespart werden.

Funktionspfeil vs. Kreuzprodukt (1)

Eine naheliegende Frage im Zshg. mit der Funktion `ersetze`:

- Warum so viele Pfeile (`->`), warum so wenige Kreuze ($\times$) in der Signatur von `ersetze`?
- Warum nicht

  "`ersetze :: (Txt` $\times$ `Vork` $\times$ `Alt` $\times$ `Neu) -> Txt`"

  statt

  `ersetze ::  Txt -> Vork -> Alt -> Neu -> Txt`?

Beachte: Das Kreuzprodukt in Haskell wird durch Tupelbeistrich ausgedrückt, d.h. "," statt $\times$. Die korrekte Haskell-Spezifikation für die Kreuzproduktvariante lautete daher:

```
ersetze :: (Txt,Vork,Alt,Neu) -> Txt
```

Funktionspfeil vs. Kreuzprodukt (2)

Beide Formen

- sind möglich, sinnvoll und berechtigt.

Funktionspfeil

- führt jedoch zu höherer (Anwendungs-) Flexibilität als Kreuzprodukt, da partielle Auswertung von Funktionen möglich ist.
- ist daher in funktionaler Programmierung die weitaus häufiger verwendete Form.

Zur Illustration:

- Berechnung der Binomialkoeffizienten.

Funktionspfeil vs. Kreuzprodukt (3)

Vergleiche die Funktionspfeilform:

```
binom :: Integer -> Integer -> Integer
binom n k
 | k==0 || n==k = 1
 | otherwise    = binom (n-1) (k-1) + binom (n-1) k
```

...mit der Kreuzproduktform:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)
 | k==0 || n==k = 1
 | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

Funktionspfeil vs. Kreuzprodukt (4)

Die höhere Flexibilität der Funktionspfeilform zeigt sich in der Anwendungssituation:

Der Funktionsterm `(binom 45)`

- ist von funktionalem Typ `(Integer -> Integer)`, eine Funktion, die ganze Zahlen in sich abbildet.
- liefert angewendet auf eine natürliche Zahl k die Anzahl der Möglichkeiten, auf die man k Elemente aus einer 45-elementigen Grundgesamtheit herausgreifen kann:

  (`(binom 45)` entspricht der Funktion "k_aus_45")

Funktionspfeil vs. Kreuzprodukt (5)

Wir können den Funktionsterm `(binom 45)` deshalb auch benutzen, um in argumentfreier Weise eine neue Funktion zu definieren, z.B. die Funktion k_aus_45 (vgl. Kapitel 1.1.1):

```
k_aus_45 :: Integer -> Integer
k_aus_45 = binom 45 -- arg.frei: k_aus_45 ist nicht
                    -- von einem Arg. gefolgt
```

Die Funktion `k_aus_45` und der Funktionsterm `(binom 45)` bezeichnen dieselbe Funktion; sie sind "Synonyme".

Aufrufe folgender Form sind deshalb möglich:

```
(binom 45) 6 ->> 8.145.060
 binom 45 6  ->> 8.145.060  -- Klammereinsparungsr.
 k_aus_45 6  ->> binom 45 6 ->> 8.145.060
```

Funktionspfeil vs. Kreuzprodukt (6)

Beachte: Auch die Funktion

```
binom' :: (Integer,Integer) -> Integer
```

ist im Haskell-Sinn einstellig.

In folgender Schreibweise wird dies besonders deutlich:

```
type IntPair = (Integer,Integer)

binom' :: IntPair -> Integer -- 1 Argument: 1-stellig
binom' p
 | snd(p) == 0 || fst(p)==snd(p) = 1
 | otherwise = binom' (fst(p)-1,snd(p)-1)
                 + binom' (fst(p)-1,snd(p))
```

`p` vom Typ `IntPair`, das eine Argument von `binom'` ist von einem Paartyp.

Funktionspfeil vs. Kreuzprodukt (7)

Beachte: binom' bietet nicht die Flexibilität von binom:

- binom' konsumiert ihr eines Argument p vom Paartyp `(Integer,Integer)` und liefert unmittelbar ein Resultat vom elementaren Typ `Integer`.

  ```
  binom' (45,6) ->> 8.145.060 :: Integer
  ```

- ein funktionales Zwischenresultat entsteht anders als bei binom nicht.

- Eine lediglich "partielle" Versorgung mit Argumenten und damit partielle Auswertung von binom' ist nicht möglich.

  Aufrufe der Form

  ```
  binom' 45
  ```

  sind syntaktisch inkorrekt und führen auf Fehlermeldungen.

Vordef. arithmetische Operationen in Pfeilform

Auch die arithmetischen (und viele weitere) Operationen sind in Haskell aus diesem Grund in der Funktionspfeilform vordefiniert:

```
(+) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
...
```

Nachstehend instantiiert für den Typ `Int`:

```
(+) :: Int -> Int -> Int
(*) :: Int -> Int -> Int
(-) :: Int -> Int -> Int
...
```

Funktionsstelligkeiten: Mathematik vs. Haskell

...unterschiedliche Sichtweisen und Akzentsetzungen.

Mathematik: Betonung der "Teile"; eine Funktion der Form

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

wird zweistellig angesehen: $\binom{\cdot}{\cdot} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

Allgemein: $f : M_1 \times \ldots \times M_n \rightarrow M$ hat Stelligkeit n.

Haskell: Betonung des "Ganzen"; eine Funktion der Form

```
type I = Integer
binom' (n,k)
  | k==0 || n==k = 1
  | otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

wird einstellig angesehen: `binom' :: (I,I) -> I`.

Zusammenfassung (1)

Für Haskell gilt:

Die Klammerung unvollständig geklammerter

- Funktionssignaturen ist rechtsassoziativ
- Funktionsterme ist linksassoziativ

zu vervollständigen.

Funktionen sind

- einstellig; sie konsumieren stets ein Argument zur Zeit.

Argumente und Werte von Funktionen und Funktionstermen

- können elementaren, zusammengesetzten oder funktionalen Typs sein.

Zusammenfassung (2)

Klammern in Signaturen und Funktionstermen

- sind mehr als schmückendes Beiwerk; sie bestimmen die Bedeutung.

Wann immer eine von den Klammereinsparungsregeln induzierte abweichende Argument- oder/und Resultatstruktur gewollt ist, muss dies durch

- explizite Klammerung in Signatur und Funktionsterm

ausgedrückt werden.

Kapitel 3.3

Curryfizierte, uncurryfizierte Funktionen

Scharf oder mild

...curryfiziert oder uncurryfiziert, das ist hier die Frage.

Curryfiziert und uncurryfiziert

...bezeichnen bestimmte ineinander überführbare Deklarationsweisen von Funktionen.

Entscheidend für die Unterscheidung ist die

- Art der Konsumation der Argumente.

Erfolgt die Konsumation

- Argument für Argument einzeln: curryfiziert
- als Tupel alle auf einmal: uncurryfiziert

Implizit liefert dies eine Unterscheidung in

- curryfizierte Funktionen
- uncurryfizierte Funktionen

Curryfiziert vs. uncurryfiziert deklariert (1)

...anhand eines Beispiels:

- `binom :: Integer -> Integer -> Integer`

  ...ist curryfiziert deklariert.
- `binom' :: (Integer,Integer) -> Integer`

  ...ist uncurryfiziert deklariert.

Curryfiziert vs. uncurryfiziert deklariert (2)

Curryfiziert deklariertes `binom`:

```
binom :: Integer -> Integer -> Integer
binom n k
 | k==0 || n==k = 1
 | otherwise = binom (n-1) (k-1) + binom (n-1) k
```

```
binom 45 6 ->> (binom 45) 6 ->> 8.145.060
           :: Integer -> Integer
                :: Integer
```

Uncurryfiziert deklariertes `binom'`:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)
 | k==0 || n==k = 1
 | otherwise = binom' (n-1,k-1) + binom' (n-1,k)
```

```
binom' (45,6) ->> 8.145.060
 :: Integer
```

Informell

Curryfizieren ersetzt

- Produkt-/Tupelbildung "$\times$" durch Funktionspfeil "$\rightarrow$".

Uncurryfizieren ersetzt

- Funktionspfeil "$\rightarrow$" durch Produkt-/Tupelbildung "$\times$".

Bemerkung: Die Bezeichnung erinnert an Haskell B. Curry; die Idee selbst ist weit älter und geht auf Moses Schönfinkel aus der Mitte der 1920er-Jahre zurück.

Die Funktionale curry und uncurry

..als Mittler zwischen curryfizierter und uncurryfizierter Darstellung:

Das Funktional curry:

$$\texttt{curry :: } \underbrace{\texttt{((a,b) -> c)}}_{\substack{\texttt{Argumenttyp von curry:}\\ \texttt{uncurryfiziert!}}} \quad \texttt{->} \quad \underbrace{\texttt{(a -> b -> c)}}_{\substack{\texttt{Resultattyp von curry:}\\ \texttt{curryfiziert!}}}$$

Das Funktional uncurry:

$$\texttt{uncurry :: } \underbrace{\texttt{(a -> b -> c)}}_{\substack{\texttt{Argumenttyp von uncurry:}\\ \texttt{curryfiziert!}}} \quad \texttt{->} \quad \underbrace{\texttt{((a,b) -> c)}}_{\substack{\texttt{Resultattyp von uncurry:}\\ \texttt{uncurryfiziert!}}}$$

Die Implementierungen v. curry u. uncurry (1)

Das Funktional curry:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)    -- x, y wird zu (x,y)
                         -- zusammengesetzt und so
                         -- für f verarbeitbar
```

Das Funktional uncurry:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y    -- (x,y) wird in x, y
                           -- getrennt und so
                           -- für g verarbeitbar
```

Die Implementierungen v. curry u. uncurry (2)

...in anderem Detail:

Das Funktional curry:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

$$\underbrace{\texttt{curry f}}_{::\ \texttt{((a,b) -> c)}}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\ \underbrace{\texttt{y}}_{::\ \texttt{b}} = \underbrace{\texttt{f}\ \underbrace{\texttt{(x,y)}}_{::\ \texttt{(a,b)}}}_{::\ \texttt{c}}$$

Das Funktional uncurry:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y
```

$$\underbrace{\texttt{uncurry g}}_{::\ \texttt{(a -> b -> c)}}\ \underbrace{\texttt{(x,y)}}_{::\ \texttt{(a,b)}} = \underbrace{\texttt{g}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\ \underbrace{\texttt{y}}_{::\ \texttt{b}}}_{::\ \texttt{c}}$$

Zur Klammerung von curry und uncurry

...vollständig, aber nicht überflüssig geklammert:

```
curry :: (((a,b) -> c) -> (a -> (b -> c)))
curry f x y  = f (x,y)

uncurry :: ((a -> (b -> c)) -> ((a,b) -> c))
uncurry g (x,y)  =  g x y
```

...miminal geklammert gemäß Klammereinsparungsregeln:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y  = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g (x,y)  =  g x y
```

curry: Schritt für Schritt zur Definition

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
curry f x y = f (x,y)
```

Sei `f` eine Funktion mit Signatur

```
f :: ((a,b) -> c)
```

Mit `f` erhalten wir für die Signaturen der Funktionsterme:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
(curry f) :: (a -> (b -> c))
((curry f) x) :: (b -> c)
(((curry f) x) y) :: c
```

Festzulegen bleibt noch der Wert des Funktionsterms:

```
(((curry f) x) y) = f (x,y) :: c
```

Nach Einsparung von Klammern erhalten wir insgesamt:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
curry f x y = f (x,y)
```

Anwendung von curry

Sei `f` uncurryfiziert gegebene Funktion mit Signatur

```
f :: ((a,b) -> c)
```

Definiere

```
g :: (a -> (b -> c))
g = curry f                          -- argumentfrei!
```

Damit

```
f (x,y) = g x y  = (curry f) x y = curry f x y
```

Übungsaufgabe 3.3.1

Vollziehe die Schritt-für-Schritt-Entwicklung zur Definition und Anwendung von `curry` für `uncurry` nach, d.h. entwickle nach dem Beispiel von `curry` Schritt für Schritt die Definition und Anwendung von `uncurry`.

Die Funktionale curry und uncurry

...bilden

- uncurryfizierte Funktionen auf ihre curryfizierten Gegenstücke ab:

  Für uncurryfiziertes `f :: (a,b) -> c` ist

  `curry f :: a -> (b -> c)`

  curryfiziert (entsprechend `g :: a -> (b -> c)`).

- curryfizierte Funktionen auf ihre uncurryfizierten Gegenstücke ab:

  Für curryfiziertes `g :: a -> (b -> c)` ist

  `uncurry g :: (a,b) -> c`

  decurryfiziert (entsprechend `f :: (a,b) -> c`).

Anwendungen von curry und uncurry

Betrachte

```
binom  :: Integer -> Integer -> Integer
binom' :: (Integer,Integer) -> Integer
```

und

```
curry   :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```

Anwendung von curry und uncurry liefert:

```
curry binom' :: Integer -> Integer -> Integer
uncurry binom :: (Integer,Integer) -> Integer
```

Somit sind folgende Aufrufe möglich und gültig:

```
curry binom' 45 6     ->> binom' (45,6) ->> 8.145.060
uncurry binom (45,6) ->> binom 45 6   ->> 8.145.060
```

Curryfiziert oder uncurryfiziert?

...das ist die Frage.

Geschmackssache? Notationelle Spielerei?

- Allenfalls bei oberflächlicher Betrachtung:
  `f x`, `f x y`, `f x y z`,... vs. `f(x)`, `f(x,y)`, `f(x,y,z)`,...

Es gilt: Nur curryfizierte Funktionen unterstützen das

- Prinzip partieller Auswertung und damit das Prinzip:

  ⇝ Funktionen liefern Funktionen als Ergebnis!

  Beispiel: Die für das Argument `45` partiell ausgewertete Funktion `binom` liefert als Resultat eine einstellige Funktion, die Funktion `k_aus_45 :: Integer -> Integer` definiert durch `k_aus_45 = (binom 45)`.

Die Bevorzugung curryfizierter Formen ist deshalb sachlich gut begründet, vorteilhaft und in der Praxis vorherrschend.

Faustregel für Funktionsdefinitionen

...curryfiziert, wo möglich, uncurryfiziert, wo nötig.

(vgl. die Funktionen `binom` und `binom'` vom Anfang dieses Abschnitts.)

Kapitel 3.4

Operatoren, Präfix- und Infixverwendung

Operatorverwendung

Präfixverwendung

- den Operanden vorangestellt:

  Beispiele: `fac 5`, `binom (45,6)`, `reverse "desserts"`, `quickSort [4,2,1,9,3,7,5]`,...

Infixverwendung

- zwischen die Operanden gestellt:

  Beispiele: `2 + 3`, `5 * 7`, `5^3`, `4 : [3,2,1]`, `[1,2,3,4]!!2`, `[3,2,1] ++ [1,2,3]`,...

Postfixverwendung

- den Operanden nachgestellt:

  Beispiele: In Haskell keine; in der Mathematik wenige, etwa die Fakultätsfunktion "!"; regelmäßig bei Verwendung "umgekehrt polnischer Notation".

Operatorverwendung in Haskell

Präfixverwendung

- ist Regelfall, insbesondere für alle selbstdeklarierten Operatoren (d.h. selbstdeklarierte Funktionen).

  Beispiele:

  - Vordefinierte Funktionen: `div`, `reverse`, `zip`,...
  - Selbstdefinierte Funktionen: `fac`, `binom`, `quicksort`,...

Infixverwendung

- ist Regelfall für einige vordefinierte Operatoren, darunter viele arithmetische Operatoren.

  Beispiele: `2 + 3`, `5 * 7`, `5^3`, `4 : [3,2,1]`, `[1,2,3,4]!!2`, `[3,2,1] ++ [1,2,3]`,...

Erweiterte Verwendungsmöglichkeiten

...gelten für binäre Operatoren in Haskell.

Infix- und Präfixverwendung ist möglich für

- vordefinierte und selbstdefinierte Binäroperatoren.

Allgemein: Wird der Binäroperator `bop` im Regelfall als

- Präfixoperator verwendet, so kann `bop` mit Hochkommata als Infixoperator `` `bop` `` verwendet werden.

  Beispiele: ``45 `binom` 6``, ``3 `mult` 5``
  (statt standardmäßig: `binom 45 6`, `mult 3 5`)

- Infixoperator verwendet, so kann `bop` geklammert als Präfixoperator `(bop)` verwendet werden.

  Beispiele: `(+) 2 3`, `(++) [3,2,1] [1,2,3]`
  (statt standardmäßig: `2 + 3`, `[2,1] ++ [1,2]`)

Beispiel

...berechne das Maximum dreier ganzer Zahlen:

```
max :: Int -> Int -> Int -> Int
max p q r
  | (mx p q == p) && (p `mx` r == p) = p
  | (mx p q == q) && (q `mx` r == q) = q
  | otherwise                        = r

  where mx :: Int -> Int -> Int
        mx p q
         | p >= q    = p
         | otherwise = q
```

Beachte: Binäroperator `mx` wird in `max` als Präfixoperator (`mx p q`) und Infixoperator (`` p `mx` r ``) verwendet.

Weitere Beispiele

... für Infix- und Präfixverwendung von Binäroperatoren anhand einiger arithmetischer Funktionen:

- Inkrement
- Dekrement
- Halbieren
- Verdoppeln
- 10er-Inkrement
- ...

Infixverwendete Binäroperatoren

- Inkrement

```
inc :: Integer -> Integer
inc n = n + 1
```

- Dekrement

```
dec :: Integer -> Integer
dec n = n - 1
```

- Halbieren

```
hlv :: Integer -> Integer
hlv n = n `div` 2  -- Nichtstandardverwendung
```

- Verdoppeln

```
dbl :: Integer -> Integer
dbl n = 2 * n
```

- 10er-Inkrement

```
inc10 :: Integer -> Integer
inc10 n = n + 10
```

Präfixverwendete Binäroperatoren

- Inkrement

```
inc :: Integer -> Integer
inc n = (+) n 1          -- Nichtstandardverw.
```

- Dekrement

```
dec :: Integer -> Integer
dec n = (-) n 1          -- Nichtstandardverw.
```

- Halbieren

```
hlv :: Integer -> Integer
hlv n = div n 2               -- Standardverw.
```

- Verdoppeln

```
dbl :: Integer -> Integer
dbl n = (*) 2 n          -- Nichtstandardverw.
```

- 10er-Inkrement

```
inc10 :: Integer -> Integer
inc10 n = (+) n 10       -- Nichtstandardverw.
```

Punktfrei als partiell ausgewertete Funktionen

- Inkrement

```
inc :: Integer -> Integer
inc = (+) 1
```

- Eins_minus (statt Dekrement)

```
eins_minus :: Integer -> Integer
eins_minus = (-) 1     -- (-) nicht kommutativ
```

- Zwei_durch (statt Halbieren)

```
zwei_durch :: Integer -> Integer
zwei_durch = div 2     -- div nicht kommutativ
```

- Verdoppeln

```
dbl :: Integer -> Integer
dbl = (*) 2
```

- 10er-Inkrement

```
inc10 :: Integer -> Integer
inc10 = (+) 10
```

Operandenstellung und Klammerung

...führen uns zu sog. Operatorabschnitten:

- Inkrement

```
inc :: Integer -> Integer
inc = (+1)
```

- Eins_minus

```
eins_minus :: Integer -> Integer
eins_minus = (1-)
```

- Verdoppeln

```
dbl :: Integer -> Integer
dbl = (2*)
```

- Halbieren

```
hlv :: Integer -> Integer
hlv = (`div` 2)
```

Beachte die unterschiedliche Klammerung und Operandenstellung in `inc`, `eins_minus`, `dbl` und `hlv`.

Kapitel 3.5

Operatorabschnitte

Operatorabschnitte (1)

Partiell ausgewertete Binäroperatoren heißen in Haskell

- Operatorabschnitte (engl. operator sections)

Beispiele:

- `(*2)` `dbl`, die Funktion, die ihr Argument verdoppelt ($\lambda x.\, x * 2$)
- `(2*)` `dbl`, s.o. ($\lambda x.\, 2 * x$)
- `(2<)` `2_kleiner_als_x`, das Prädikat, das überprüft, ob sein Argument größer als 2 ist ($\lambda x.\, 2 < x$)
- `(<2)` `x_kleiner_als_2`, das Prädikat, das überprüft, ob sein Argument kleiner als 2 ist ($\lambda x.\, x < 2$)
- `(2:)` `headAppend`, die Funktion, die 2 an den Anfang einer typkompatiblen Liste setzt
- ...

Operatorabschnitte (2)

Beispiele (fgs.):

- `(+1)`, `(1+)` `inc`, die Funktion, die ihr Argument um 1 erhöht $(\lambda x.\, x+1)$ bzw. $(\lambda x.\, 1+x)$
- `(1-)` `eins_minus`, die Funktion, die ihr Argument von 1 abzieht $(\lambda x.\, 1-x)$
- `(-1)` kein Operatorabschn., sondern d. Zahl '-1'.
- ``(`div` 2)`` `hlv`, die Funktion, die ihr Argument ganzzahlig halbiert $(\lambda x.\, x\ \mathit{div}\ 2)$
- ``(2 `div`)`` `zwei_durch`, die Funktion, die 2 ganzzahlig durch ihr Argument teilt $(\lambda x.\, 2\ \mathit{div}\ x)$
- ...
- `(div 2)`, `div 2` `zwei_durch`, s.o. $(\lambda x.\, 2\ \mathit{div}\ x)$; keine echten Operatorabschnitte, sondern gewöhnliche Präfixoperatorverwendung.

Operatorabschnitte (3)

Operatorabschnitte können in Haskell gebildet werden mit

- vordefinierten und selbstdefinierten binären Operatoren.

Beispiele für die curryfizierte Funktion `binom` (vgl. Kapitel 3.2):

- `(binom 45)` 45_über_k, die Funktion "k_aus_45".
- ``(45 `binom`)`` 45_über_k, s.o.
- ``(`binom` 6)`` n_über_6, die Funktion "6_aus_n".
- ...

Beachte: Mit der uncurryifzierten Funktion `binom'` (vgl. Kapitel 3.2) können keine Operatorabschnitte gebildet werden.

Anwendung: Punktfreie, argumentlose

...Funktionsdefinitionen mit Operatorabschnitten:

- "45_über_k" bzw. "k_aus_45"

```
k_aus_45 :: Integer -> Integer
k_aus_45 = binom 45
```

```
k_aus_45 :: Integer -> Integer
k_aus_45 = (45 `binom`)
```

- "n_über_6" bzw. "6_aus_n"

```
sechs_aus_n :: Integer -> Integer
sechs_aus_n = (`binom` 6)
```

- Inkrement

```
inc :: Integer -> Integer
inc = (+1)
```

- Verdoppeln

```
dbl :: Integer -> Integer
dbl = (2*)
```

- ...

Nichtkommutative Operatoren

...benötigen Obacht bei der Bildung von Operatorabschnitten.

Infix- und Präfixbenutzung hat einen Bedeutungsunterschied. Am Beispiel von `div`:

- Infixverwendung führt zu den Funkionen `hlv` und `zwei_durch`.
- Präfixverwendung führt zur Funktion `zwei_durch`.

bei ansonsten gleicher partieller Auswertung.

Am Beispiel von div für hlv und zwei_durch

- Halbieren ("durch_zwei") (div infixverwendet)

```
hlv :: Integer -> Integer
hlv = (`div` 2)            -- Operatorabschnitt

hlv 5 ->> 2, hlv 10 ->> 5, hlv 15 ->> 7
```

- "zwei_durch" (div präfixverwendet)

```
zwei_durch :: Integer -> Integer
zwei_durch = div 2          -- Präfixverwendung
-- bedeutungsgleich mit:
zwei_durch = (2 `div`)    -- Operatorabschnitt

zwei_durch 5 ->> 0, zwei_durch 2 ->> 1,
zwei_durch 1 ->> 2
```

Am Beispiel von (-) für `eins_minus`

- "eins_minus" ((-) infixverwendet)

```
eins_minus :: Integer -> Integer
eins_minus = (1-)            -- Operatorabschnitt

eins_minus 5    ->> -4, eins_minus 1 ->> 0,
eins_minus (-1) ->> 2
```

- "eins_minus" ((-) präfixverwendet)

```
eins_minus :: Integer -> Integer
eins_minus = (-) 1            -- Präfixverwendung
 -- bedeutungsgleich mit:
eins_minus = (1-)            -- Operatorabschnitt
```

Beachte: (-1) repräsentiert die Zahl '-1', keinen Operatorabschnitt.

Am Beispiel von (+) für `inc`

...kein Unterschied wg. Kommutativität der Addition:

- Inkrement ((+) infixverwendet)

```
inc :: Integer -> Integer
inc = (+1)                  -- Operatorabschnitt
-- bedeutungsgleich mit:
inc = (1+)                  -- Operatorabschnitt
```

```
inc 5 ->> 6, inc 10 ->> 11, inc 15 ->> 16
```

- Inkrement ((+) präfixverwendet)

```
inc :: Integer -> Integer
inc = (+) 1                  -- Präfixverwendung
-- entspricht:
inc = (1+)
```

```
inc 5 ->> 6, inc 10 ->> 11, inc 15 ->> 16
```

Am Beispiel von (∗) für dbl

...kein Unterschied wg. Kommutativität der Multiplikation:

- Verdoppeln ((∗) infixverwendet)

```
dbl :: Integer -> Integer
dbl = (*2)                    -- Operatorabschnitt
-- bedeutungsgleich mit:
dbl = (2*)                    -- Operatorabschnitt

dbl 5 ->> 10, dbl 10 ->> 20, dbl 15 ->> 30
```

- Verdoppeln ((∗) präfixverwendet)

```
dbl :: Integer -> Integer
dbl = (*) 2                    -- Präfixverwendung
-- entspricht:
dbl = (2*)

dbl 5 ->> 10, dbl 10 ->> 20, dbl 15 ->> 30
```

Zusammenfassung

Ist op ein Binäroperator und sind x und y typgeeignete Operanden für op, dann heißen die Ausdrücke

- (op), (x op), (op y)

Operatorabschnitte, die für folgende Funktionen stehen:

- (op) = (λ x. (λ y. x op y))
- (x op) = (λ y. x op y)
- (op y) = (λ x. x op y)

und somit besonders knappe Funktionsdefinitionen erlauben (Sonderfall: Der Subtraktionsoperator (-)).

Kapitel 3.6

Angemessene, unangemessene Funktionsdefinitionen

Total vs. partiell definierte Funktionen (1)

Total definierte Funktionen sind die Ausnahme, partiell definierte Funktionen die Regel.

Betrachte z.B. die Funktionen `fac` und `fib`, deren Auswertung nur für nichtnegative Argumente terminiert und definiert ist:

```
fac :: Int-> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Total vs. partiell definierte Funktionen (2)

...als Implementierungen der (auf den natürlichen Zahlen total definierten) Fakultäts- und Fibonacci-Funktion:

$$! : \mathbb{N} \rightarrow \mathbb{N}$$

$$n\,! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{sonst} \end{cases}$$

$$fib : \mathbb{N} \rightarrow \mathbb{N}$$

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n-2) + fib(n-1) & \text{sonst} \end{cases}$$

Transparenz der Partialität

Partialität der Implementierungen von fac und fib

- i.w. technisch induziert (Abwesenheit eines Datentyps für natürliche Zahlen).

Explizite, transparente Sichtbarmachung der Partialität ist dennoch sinnvoll, angemessen und auch einfach möglich:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | n >= 1 = n * fac (n - 1)
      | otherwise = error "undefiniert"

fib :: Int -> Int
fib n | n == 0 = 1
      | n == 1 = 1
      | n >= 2 = fib (n-2) + fib (n-1)
      | otherwise = error "undefiniert"
```

Partiell definierte Funktionen

...sind auch `f`, `g` und `h`:

$$f \;: \mathbb{Z} \to \mathbb{Z}$$

$$f(z) = \begin{cases} 2 & \text{falls } z \geq 1 \\ \mathit{undef} & \text{sonst} \end{cases}$$

$$g \;: \mathbb{Z} \to \mathbb{Z}$$

$$g(z) = \begin{cases} 2^z & \text{falls } z \geq 1 \\ \mathit{undef} & \text{sonst} \end{cases}$$

$$h \;: \mathbb{Z} \to \mathbb{Z}$$

$$h(z) = \begin{cases} 2^1 & \text{falls } z = 1 \\ 2^{(|z|+2)} & \text{falls } z \leq 0 \\ \mathit{undef} & \text{sonst} \end{cases}$$

Angemessene Implementierungen von f, g, h

...die Partialität der Implementierungen von f, g und h liegt transparent und offen zutage:

```
f :: Integer -> Integer
f z | z >= 1    = 2
    | otherwise = error "undefiniert"

g :: Integer -> Integer
g z | z >= 1    = 2^z
    | otherwise = error "undefiniert"

h :: Integer -> Integer
h z | z == 1    = 2
    | z <= 0    = 2^((abs z)+2)
    | otherwise = error "undefiniert"
```

Unangemessene Implementierung von f

Betrachte folgende intransparente Implementierung von f:

```
f :: Integer -> Integer
f 1 = 2
f x = 2 * (f x)
```

Verschleiernd und intransparent, auch wenn man sich durch Nachrechnen darüber versichern kann:

Die Auswertung von f terminiert für $n = 1$:

```
f 1 ->> 2
```

...und für keinen von 1 verschiedenen Argumentwert.

Auswertungsbeispiele für `f`

Die Auswertung von `f` terminiert für $n = 1$:

```
f 1    ->> 2
```

Die Auswertung von `f` terminiert nicht für $n \neq 1$:

```
f (-9) ->> 2 * (f (-9)) ->> 2 * (2 * (f (-9)))
       ->> 2 * (2 * (2 * (f (-9)))) ->>...
f (-1) ->> 2 * (f (-1)) ->> 2 * (2 * (f (-1)))
       ->> 2 * (2 * (2 * (f (-1)))) ->>...
f 0    ->> 2 * (f 0)    ->> 2 * (2 * (f 0))
       ->> 2 * (2 * (2 * (f 0)))    ->>...
f 2    ->> 2 * (f 2)    ->> 2 * (2 * (f 2))
       ->> 2 * (2 * (2 * (f 2)))    ->>...
f 3    ->> 2 * (f 3)    ->> 2 * (2 * (f 3))
       ->> 2 * (2 * (2 * (f 3)))    ->>...
f 9    ->> 2 * (f 9)    ->> 2 * (2 * (f 9))
       ->> 2 * (2 * (2 * (f 9)))    ->>...
```

Angemessenheit vs. Unangemessenheit für f

...beide Implementierungen der Funktion f sind formal und inhaltlich korrekt, dennoch:

Unangemessen, weil intransparent und verschleiernd, ist:

```
f :: Integer -> Integer
f 1 = 2
f x = 2 * (f x)
```

Angemessen, weil transparent und offen legend, ist:

```
f :: Integer -> Integer
f z | z = 1     = 2
    | otherwise = error "undefiniert"
```

Unangemessene Implementierungen von g, h

Betrachte folgende intransparente Implementierungen von g und h:

```
g :: Integer -> Integer
g 1     = 2
g (x+1) = 2 * (g x)

h :: Integer -> Integer
h 1 = 2
h x = 2 * (h (x+1))
```

Bemerkung: Muster der Form (x+1) wie in der Definition von g nicht mehr zulässig in neueren Haskell-Versionen.

Auswertungsbeispiele für g

Die Ausw. von g terminiert für echt positive Argumentwerte:

```
g 1 ->> 2
g 2 ->> g (1+1) ->> 2 * (g 1) ->> 2 * 2 ->> 4
g 3 ->> g (2+1) ->> 2 * (g 2) ->> 2 * g (1+1)
    ->> 2 * (2 * (g 1)) ->> 2 * (2 * 2) ->> 2 * 4
    ->> 8
g 9 ->> g (8+1) ->> 2 * (2 * (g 8)) ->> ... ->> 512
```

Die Auswertung von g terminiert nicht sonst:

```
g 0      ->> g ((-1)+1)        ->> 2 * (g (-1))
         ->> 2 * (g ((-2)+1)) ->> 2 * (2 * (g (-2)))
         ->> 2 * (2 * (g ((-3)+1)))
         ->> 2 * (2 * (2 * (g (-3)))) ->>...
g (-1)   ->> g ((-2)+1)        ->> 2 * (g (-2))  ->>...
g (-9)   ->> g ((-10)+1)       ->> 2 * (g (-10)) ->>...
```

Auswertungsbeispiele für h

Die Auswertung von h terminiert für Argumentwerte ≤ 1:

```
h 1    ->> 2
h 0    ->> 2 * (h (0+1)) ->> 2 * (h 1) ->> 2 * 2
       ->> 4
h (-1) ->> 2 * (h ((-1)+1)) ->> 2 * (h 0)
       ->> ... ->> 2 * 4 ->> 8
h (-9) ->> 2 * (h ((-9)+1))
       ->> 2 * (h (-8)) ->> ... ->> 2048
```

Die Auswertung von h terminiert nicht sonst:

```
h 2 ->> 2 * (h (2+1)) ->> 2 * (h 3)
    ->> 2 * (2 * (h (3+1))) ->> 2 * (2* (h 4)) ->>...
h 3 ->> 2 * (h (3+1)) ->> 2 * (h 4)  ->>...
h 9 ->> 2 * (h (9+1)) ->> 2 * (h 10) ->>...
```

Angemessenheit vs. Unangemessenheit für g, h

...beide Implementierungen der Funktionen g und h sind formal und inhaltlich korrekt, dennoch:

Unangemessen, weil intransparent und verschleiernd, ist:

```
g :: Integer -> Integer
g 1     = 2
g (x+1) = 2 * (g x)
```

```
h :: Integer -> Integer
h 1 = 2
h x = 2 * (h (x+1))
```

Angemessen, weil transparent und offen legend, ist:

```
g :: Integer -> Integer
g z
 | z >= 1 = 2^z
 | otherwise
    = error "undefiniert"
```

```
h :: Integer -> Integer
h z
 | z == 1 = 2
 | z <= 0 = 2^((abs z)+2)
 | otherwise
    = error "undefiniert"
```

Kapitel 3.7

Funktions- und Programmlayout, Abseitsregel

Programmlayout, Programmbedeutung

Für die meisten Programmiersprachen gilt:

- Das Layout des Programmtexts beeinflusst
  - seine Lesbarkeit, Verständlichkeit, Wartbarkeit
  - aber nicht seine Bedeutung

Nicht so für Haskell – für Haskell gilt:

- Das Layout des Programmtexts trägt Bedeutung!

Dieser Aspekt des Sprachentwurfs

- ist für Haskell grundsätzlich anders entschieden worden als für Sprachen wie Java, Pascal, C und viele andere.
- ersetzt begin/end- bzw. {/}-Konstrukte durch Layoutanforderungen.
- kann als Reminiszenz an Sprachen wie Cobol, Fortran gesehen werden, findet sich aber auch in anderen modernen Sprachen wie z.B. occam.

Bindungs- und Gültigkeitsbereiche in Haskell

...bestimmt durch layoutabhängige Syntax.

Eröffnung, Fortsetzung und Beendigung eines Bindungs- und Gültigkeitsbereichs (Jargon: "Box") gemäß "Abseits"-Regel:

- Das jeweils erste Zeichen einer Deklaration (auch nach `let`, `where`) eröffnet eine neuen Bereich.
- Ist die nächste Zeile
  - gegenüber der aktuellen Box nach rechts eingerückt:
    $\rightsquigarrow$ die aktuelle Zeile wird fortgesetzt
  - genau am linken Rand der aktuellen Box:
    $\rightsquigarrow$ eine neue Deklaration wird eingeleitet
  - weiter links als die aktuelle Box:
    $\rightsquigarrow$ die aktuelle Box wird beendet ("Abseitssituation")

Veranschaulichung anhand einer Funktion `kugelOV` zur Berechnung von Oberfläche und Volumen einer Kugel mit Radius r: Oberfläche: $4\pi r^2$; Volumen: $\frac{4}{3}\pi r^3$.

kugelOV "üblich" ausgelegt

```
type Radius      = Float
type Oberflaeche = Float
type Volumen     = Float

pi = 3.14

kugelOV :: Radius -> (Oberflaeche,Volumen)
kugelOV r = (oberflaeche r,volumen r)
 where oberflaeche :: Radius -> Oberflaeche
       oberflaeche r = 4 * pi * square r

       volumen :: Radius -> Volumen
       volumen r = (4/3) * pi * cubic r
        where cubic x = x * square x

square :: Float -> Float
square x = x^2
```

kugelOV korrekt, aber “unschön” ausgelegt

```
type Radius      = Float
type Oberflaeche = Float
type Volumen     = Float

pi = 3.14 :: Float

kugelOV :: Radius -> (Oberflaeche,Volumen)
kugelOV r =
 (oberflaeche r,
   volumen r)
    where oberflaeche :: Radius -> Oberflaeche
          oberflaeche r = 4 * pi *
              square r
          volumen :: Radius -> Volumen
          volumen r = (4/3)
               * pi * cubic r
              where cubic x = x * square x

square :: Float -> Float
square x = x^2
```

Graphische Veranschaulichung des Box-Begriffs

```
type Radius      = Float -- Radius, Oberflaeche, Volumen und pi
type Oberflaeche = Float -- global im Gesamtprogramm sichtbar
type Volumen     = Float

pi = 3.14 :: Float
-----------------------------------------------
| -- kugelOV global im Gesamtprogramm sichtbar
kugelOV :: Radius -> (Oberflaeche,Volumen)
kugelOV =
|(oberflaeche r,
|  volumen r)
|         --------------------------------
|         | -- oberflaeche lokal in kugelOV sichtbar
|    where oberflaeche :: Radius -> Oberflaeche
|          oberflaeche r = 4 * pi *
|          |   square r
|          ------------------------->
|          --------------------------------
|          | -- volumen lokal in kugelOV sichtbar
|          volumen :: Radius -> Volumen
|          volumen r = (4/3)
|          |    * pi * cubic r
|          | ------------------------
|          | | -- cubic lokal in volumen sichtbar
|          | | where cubic x = x * square x
|          | ------------------->
|          ----------------------->
------------------------------------>
-----------------------------------------------
| -- square global im Gesamtprogramm sichtbar
square :: Float -> Float
square x = x^2
------------------------------------>
```

Bewährte Layoutkonventionen (1)

...zur Einhaltung der Abseitsregel für Funktionsdefinitionen:

```
funktionsName parameter_1 parameter_2... parameter_n
 | waechter_1 = ausdruck_1
 | waechter_2 = ausdruck_2
 ...
 | otherwise  = ausdruck_k
   where
   v_1 a_1 ... a_n = r_1    -- v_1, v_2,..., sichtbar
   v_2             = r_2    -- in der gesamten Funk-
   ...                      -- tion funktionsName,
                            -- aber nicht außerhalb.
```

Bewährte Layoutkonventionen (2)

...zur Umgang mit langen Bedingungen und Ausdrücken:

```
funktionsName parameter_1 parameter_2... parameter_n
  | waechter_1 = ausdruck_1
  | waechter_2 = ausdruck_2
  | diesIsteineGanz
    BesondersLangeMehrzeilige
    BedingungAlsWaechter
              = diesIstEinBesonders
                LangerMehrzeiliger
                AusdruckZurWertfestlegung
  | waechter_4 = ausdruck_4
  ...
  | otherwise  = ausdruck_k
    where...
```

Sprachkonstrukt- und Layoutwahl

...nach Angemessenheitserwägungen:

- Was ist gut und einfach lesbar und verständlich?

Illustration:

Vergleiche folgende je drei Implementierungen der Rechenvorschriften

- `fib :: Int -> Int`
- `max :: Int -> Int -> Int -> Int`

Drei Implementierungen für `fib`

Mittels Muster:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Mittels bedingter Ausdrücke:

```
fib :: Int -> Int
fib n = if (n == 0) || (n == 1) then 1
             else fib (n-2) + fib (n-1)
```

Mittels geschachtelter bedingter Ausdrücke:

```
fib :: Int -> Int
fib n = if n == 0
             then 1
             else if n == 1
                   then 1
                   else fib (n-2) + fib (n-1)
```

Drei Implementierungen für max

Mittels geschachtelter bedingter Ausdrücke und anonymer λ-Abstraktion:

```
max :: Int -> Int -> Int -> Int
max= \p q r -> if p>=q then (if p>=r then p else r)
                       else (if q>=r then q else r)
```

Mittels geschachtelter bedingter Ausdrücke:

```
max :: Int -> Int -> Int -> Int
max p q r = if (p>=q) && (p>=r) then p
            else if (q>=p) && (q>=r) then q else r
```

Mittels bewachter Ausdrücke:

```
max :: Int -> Int -> Int -> Int
max p q r
 | (p>=q) && (p>=r)  = p
 | (q>=p) && (q>=r)  = q
 | otherwise         = r
```

Richtschnur

Programme können grundsätzlich
auf zwei Arten geschrieben werden:
So einfach, dass sie offensichtlich keinen Fehler enthalten,
So kompliziert, dass sie keinen offensichtlichen Fehler enthalten.

C.A.R. "Tony" Hoare
Turing Award Preisträger

- Angemessen gewählte Sprachkonstrukte und gutes Layout unterstützen dabei, einfache und offensichtlich fehlerfreie Programme zu schreiben (vgl. Kapitel 3.6)!
- In Haskell heißt dies "schönes" Einrücken und zumeist Verwendung bewachter Ausdrücke und Muster anstelle (geschachtelter) bedingter Ausdrücke.

Kapitel 3.8

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 3 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 3, Funktionen und Operatoren; Kapitel 4, Rekursion als Entwurfstechnik)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 1, Elemente funktionaler Programmierung)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3.5, Function types; Kapitel 3.6, Curried functions; Kapitel 4, Defining functions; Kapitel 6, Recursive functions)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 3 (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions; Kapitel 4, Hello Recursion!)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming - Partial Function Application and Currying)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 6, Ein bisschen syntaktischer Zucker)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 7, Defining functions over lists)

Kapitel 4

Typsynonyme, Neue Typen, Typklassen

Kapitel 4.1

Typsynonyme

Typsynomyme

...oder: Was bedeuten unsere Daten?

Was bedeuten unsere Daten?

Werte und ihre Typinformation allein erlauben oft nur wenig oder keinen Aufschluss darüber, was für Daten sie modellieren oder repräsentieren. Betrachte:

```
('A',True)  :: (Char,Bool)
('Z',False) :: (Char,Bool)

("Fun",3)   :: (String,Int)
("Hello",5) :: (String,Int)

(5.0,8.0,6.5) :: (Float,Float,Float)
(7.2,9.6,8.4) :: (Float,Float,Float)

[(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
                :: [(Float,Float,Float)]
[(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)],
 (6.3,7.7,7.9)] :: [(Float,Float,Float)]
```

Sprechende Funktionsnamen

...in Anwendungen können helfen, die äußere Semantik der modellierten Daten aufzudecken.

```
erstesZeichen :: String -> (Char,Bool)
erstesZeichen "" = error "Fehler: Arg. ist leeres Wort"
erstesZeichen (c:_) = (c, elem c ['A','E','I,'O','U'])

erstesZeichen "Alpha" ->> ('A',True)
erstesZeichen "Zeta"  ->> ('Z',False)

echoLaenge :: String -> (String,Int)
echoLaenge s = (s,length s)

echoLaenge "Fun"   ->> ("Fun",3)
echoLaenge "Hello" ->> ("Hello",5)
```

Aber nicht immer

...oder nicht immer vollständig:

```
auswertung :: (Float,Float) -> (Float,Float,Float)
auswertung (x,y) = (x,y,arithMittel)
 where arithMittel = (x+y) / 2

auswertung (5.0,8.0) ->> (5.0,8.0,6.5)
auswertung (7.2,9.6) ->> (7.2,9.6,8.4)

reihenausw :: [(Float,Float)] -> [(Float,Float,Float)]
reihenausw [] = []
reihenausw ((x,y) : xys)
             = (auswertung (x,y)) : reihenausw xys

reihenausw [(5.0,8.0),(7.2,9.4),(1.5,4.1)]
  ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw [(2.4,7.8),(3.2,5.4),(2.5,8.1)],(6.3,7.7)]
  ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3),
       (6.3,7.7,7.9)]
```

Die äußere Semantik der Gleitkommatupel bleibt verborgen.

Was bedeuten also unsere Daten?

Wofür stehen Gleitkommapaare und -tripel wie

```
(5.0,8.0)
(7.2,9.6)

(5.0,8.0,6.5)
(7.2,9.6,8.4)
```

...und wofür Listen solcher Paare und Tripel?

```
[(5.0,8.0),(7.2,9.4),(1.5,4.1)]
[(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3),
 (6.3,7.7,7.9)]
```

Für Aktienkurse, für Pegelstände, für Ortsdaten?

Typsynonyme schaffen Abhilfe: Aktienkurse

```
type Kurs                = Float
type Niedrigst           = Kurs
type Hoechst             = Kurs
type Geglaettet          = Kurs
type Kursausschlag       = (Niedrigst,Hoechst)
type Ausschlagsanalyse   = (Niedrigst,Hoechst,Geglaettet)
type Kursverlauf         = [Kursausschlag]
type Verlaufsanalyse     = [Ausschlagsanalyse]
```

Das erlaubt jetzt folgende sprechendere Funktionsdefinitionen:

```
auswertung :: Kursausschlag -> Ausschlagsanalyse
auswertung (x,y) = (x,y,arithMittel)
 where arithMittel = (x+y) / 2

reihenausw :: Kursverlauf -> Verlaufsanalyse
reihenausw [] = []
reihenausw ((x,y) : xys)
              = (auswertung (x,y)) : reihenausw xys
```

Typsynonyme schaffen Abhilfe: Aktienkurse

...und Anwendungsaufrufe:

```
ausschlag1 = (5.0,8.0) :: Kursausschlag
ausschlag2 = (7.2,9.6) :: Kursausschlag

auswertung ausschlag1 ->> (5.0,8.0,6.5)
auswertung ausschlag2 ->> (7.2,9.6,8.4)


verlauf1 = [(5.0,8.0),(7.2,9.4),(1.5,4.1)] :: Kursverlauf
verlauf2 = [(2.4,7.8),(3.2,5.4),(2.5,8.1)] :: Kursverlauf

reihenausw verlauf1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw verlauf2
 ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Typsynonyme schaffen Abhilfe: Pegelstände

```
type Pegelstand        = Float
type Niedrig           = Pegelstand
type Hoch              = Pegelstand
type Mittel            = Pegelstand
type Messung           = (Niedrig,Hoch)
type Auswertung        = (Niedrig,Hoch,Mittel)
type Messreihe         = [Messung]
type Auswertungsreihe = [Auswertung]
```

Das ermöglicht jetzt folgende Funktionsdefinitionen:

```
auswertung' :: Messung -> Auswertung
auswertung' (x,y) = (x,y,arithMittel)
 where arithMittel = (x+y) / 2

reihenausw' :: Messreihe -> Auswertungsreihe
reihenausw' [] = []
reihenausw' ((x,y) : xys)
             = (auswertung' (x,y)) : reihenausw' xys
```

Typsynonyme schaffen Abhilfe: Pegelstände

...und Anwendungsaufrufe:

```
mess1 = (5.0,8.0) :: Messung
mess2 = (7.2,9.6) :: Messung

auswertung' mess1 ->> (5.0,8.0,6.5)
auswertung' mess2 ->> (7.2,9.6,8.4)

messreihe1 = [(5.0,8.0),(7.2,9.4),(1.5,4.1)] :: Messreihe
messreihe2 = [(2.4,7.8),(3.2,5.4),(2.5,8.1)] :: Messreihe

reihenausw' messreihe1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw' messreihe2
 ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Typsynonyme schaffen Abhilfe: Ortsdaten

```
type Koordinate  = Float
type X           = Koordinate
type Y           = Koordinate
type Z           = Koordinate
type Ebenenpunkt = (X,Y)
type Raumpunkt   = (X,Y,Z)
type Flaeche     = [Ebenenpunkt]
type Koerper     = [Raumpunkt]
```

Das ermöglicht folgende Funktionsdefinitionen:

```
auswertung'' :: Ebenenpunkt -> Raumpunkt
auswertung'' (x,y) = (x,y,arithMittel)
 where arithMittel = (x+y) / 2

reihenausw'' :: Flaeche -> Koerper
reihenausw'' [] = []
reihenausw'' ((x,y) : xys)
             = (auswertung'' (x,y)) : reihenausw'' xys
```

Typsynonyme schaffen Abhilfe: Ortsdaten

...und Anwendungsaufrufe:

```
xy1 = (5.0,8.0) :: Ebenenpunkt
xy2 = (7.2,9.6) :: Ebenenpunkt

auswertung'' xy1 ->> (5.0,8.0,6.5)
auswertung'' xy2 ->> (7.2,9.6,8.4)

flaeche1 = [(5.0,8.0),(7.2,9.4),(1.5,4.1)] :: Flaeche
flaeche2 = [(2.4,7.8),(3.2,5.4),(2.5,8.1)] :: Flaeche

reihenausw'' flaeche1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw'' flaeche2
 ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Selektorfunktionen für Tupel

Selektorfunktionen oder kurz Selektoren

- für Paare vordefiniert in Haskell (`fst`, `snd`), s. Kap. 2.2.1
- für höherstellige Tupel selbst zu definieren

Generell gilt: Für Selektoren sind musterbasierte Definitionen meist am zweckmäßigsten.

Selektoren für Wertpapierdaten

......in musterbasierter Definitionsweise:

```
ndgstKurs :: Kursausschlag -> Niedrigst
ndgstKurs (ndgst,_) = ndgst

hchstKurs :: Kursausschlag -> Hoechst
hchstKurs (_,hchst) = hchst

ndgstKursA :: Ausschlagsanalyse -> Niedrigst
ndgstKursA (ndgst,_,_) = ndgst

hchstKursA :: Ausschlagsanalyse -> Hoechst
hchstKursA (_,hchst,_) = hchst

ggKursA :: Ausschlagsanalyse -> Geglaettet
ggKursA (_,_,ggk) = ggk
```

In gleicher Weise

...lassen sich Selektoren für

- Pegelstandsdaten
- Ortsdaten

definieren.

Selektoren für Pegelstandsdaten

...in musterbasierter Definitionsweise:

```
nPglM :: Messung -> Niedrig
nPglM (n,_) = n

hPglM :: Messung -> Hoch
hPglM (_,h) = h

nPglA :: Auswertung -> Niedrig
nPglA (n,_,_) = n

hPglA :: Auswertung -> Hoch
hPglA (_,h,_) = h

mPglA :: Auswertung -> Mittel
mPglA (_,_,m) = m
```

Selektoren für Ortsdaten

......in musterbasierter Definitionsweise:

```
xE :: Ebenenpunkt -> X
xE (x,_) = x
yE :: Ebenenpunkt -> Y
yE (_,y) = y

xR :: Raumpunkt -> X
xR (x,_,_) = x
yR :: Raumpunkt -> Y
yR (_,y,_) = y
zR :: Raumpunkt -> Z
zR (_,_,z) = z
```

Die Selektoren

...für Kursverläufe, Messungen und Ebenenpunkte abgestützt auf die vordefinierten Paarselektoren:

```
ndgstKurs :: Kursausschlag -> Niedrigst
ndgstKurs = fst

hchstKurs :: Kursausschlag -> Hoechst
hchstKurs = snd

nPglM :: Messung -> Niedrig
nPglM = fst

hPglM :: Messung -> Hoch
hPglM = snd

xE :: Ebenenpunkt -> X
xE = fst

yE :: Ebenenpunkt -> Y
yE = snd
```

Zwei weitere Beispiele

...für

- Studentendaten
- Buchhandelsdaten

Typsynonyme u. Selektoren f. Studentendaten

```
type Vorname          = String
type Nachname         = String
type Email            = String
type Studienkennzahl  = Int
type Skz              = Studienkennzahl
type Student          = (Vorname,Nachname,Email,Skz)

(,,,) "Max Muster" "e123456@stud.tuwien.ac.at" 534
 ->> ("Max","Muster","e123456@stud.tuwien.ac.at",534)
                                                    :: Student

vorname :: Student -> Vorname                 (Ausschließlich
vorname (v,n,e,k) = v                       Variablenmuster)

nachname :: Student -> Nachname
nachname (v,n,e,k) = n

email :: Student -> Email
email (v,n,e,k) = e

skz :: Student -> Studienkennzahl
skz (v,n,e,k) = k
```

Typsynonyme u. Selektoren f. Buchhandelsdat.

```
type Autor   = String
type Titel   = String
type Auflage = Int
type Jahr    = Int
type Lagernd = Bool
type Buch    = (Autor,Titel,Auflage,Jahr,Lagernd)

(,,,,) "Simon Thompson" "Haskell" 3 2011 True
 ->> ("Simon Thompson","Haskell",3,2011,True) :: Buch

autor :: Buch -> Autor                        (Variablenmuster
autor (a,_,_,_,_) = a                          und Wild Card)

titel :: Buch -> Titel
titel (_,t,_,_,_) = t

auflage :: Buch -> Auflage
auflage (_,_,a,_,_) = a

erschienen :: Buch -> Jahr
erschienen (_,_,_,j,_) = j

lagernd :: Buch -> Lagernd
lagernd (_,_,_,_,l) = l
```

Zusammenfassung

Typsynonyme

- erlauben die äußere Semantik von Datentypen durch Wahl eines guten und sprechenden Namens offenzulegen und mitzuteilen.
- ermöglichen damit aussagekräftigere und sprechendere Funktionssignaturen.
- erhöhen dadurch die Lesbarkeit, Verständlichkeit und Transparenz von und in Programmen.

Aber: Typsynonyme

- führen keine neuen Typen ein, sondern ausschließlich neue Namen für bereits existierende Typen, sog. Alias-Namen oder Synonyme.
- leisten daher keinen Beitrag zu mehr Typsicherheit.

Dazu mehr in Kapitel 4.2 und Kapitel 5.

Kapitel 4.2

Neue Typen

Neue Typen

Typsynonyme

- führen keine neuen Typen ein, sondern neue Namen für bereits existierende Typen, sog. Aliasnamen oder Synonyme.
- leisten daher keinen Beitrag zu mehr Typsicherheit.

Warum?

Neue Typen

Typsynonyme

- dürfen überall dort stehen und verwendet werden, wo auch ihre jeweiligen Grundtypen stehen dürfen und umgekehrt.

In anderen Worten:

Typsynonyme und ihre jeweiligen Grundtypen

- dürfen sich ohne Einschränkung wechselweise vertreten.

Datenzusammengehörigkeit

...im Sinne intendierter Typbedeutung ist vertikal ausgedrückt:

Grundtyp	Durch Typsynonym intendierter Typ		
	Wertpapierdaten	Pegeldaten	Ortsdaten
`Float`	Kurs Niedrigst Hoechst Geglaettet	Pegelstand Niedrig Hoch Mittel	Koordinate X Y Z
`(Float,Float)`	Kursausschlag (ndgst,hchst)	Messung (ndg,hch)	Ebenenpunkt (x,y)
`(Float,Float,Float)`	Ausschlagsanalyse (ndgst,hchst,gg)	Auswertung (ndg,hch,mtt)	Raumpunkt (x,y,z)
`[(Float,Float)]`	Kursverlauf Kursausschlag-L.	Messreihe Mess'gen-L.	Fläche E'punkt-Liste
`[(Float,Float,Float)]`	Verlaufsanalyse Ausschlagsa'yse-L.	Ausw'gsreihe Ausw'gs-L.	Körper R'punkt-Liste

Datenzusammengehörigkeit

...im Sinne tatsächlicher Typbedeutung ist horizontal ausgedrückt:

Die (jeweils gleichfarbigen) Typ (-bezeichner)

- `Float`, `Kurs`, `Niedrigst`, `Hoechst`, `Geglaettet`, `Pegelstand`, `Niedrig`, `Hoch`, `Mittel`, `Koordinate`, `X`, `Y`, `Z`
- `(Float,Float)`, `Kursausschlag`, `Messung`, `Ebenenpunkt`
- `(Float,Float,Float)`, `Ausschlagsanalyse`, `Auswertung`, `Raumpunkt`
- [(Float,Float)], Kursverlauf, Messreihe, Flaeche
- [(Float,Float,Float)], Verlaufsanalyse, Auswertungsreihe, Koerper

...dürfen einander (im Widerspruch zu ihrer intendierten Bedeutung) wechselweise vertreten.

Typisierung aus Haskell-Sicht (1)

Für die Typisierung gilt deshalb (trotz oder wegen der Verwendung von Typsynoymen):

```
Kurs, Niedrigst, Hoechst, Geglaettet,
Pegelstand, Hoch, Niedrig, Mittel,
Koordinate, X, Y, Z                      :: Float

Kursausschlag, Messung, Ebenenpunkt  :: (Float,Float)

Ausschlagsanalyse, Auswertung, Raumpunkt
                           :: (Float,Float,Float)

Kursverlauf, Messreihe, Flaeche :: [(Float,Float)]

Verlaufsanalyse, Auswertungsreihe, Koerper
                         :: [(Float,Float,Float)]
```

Typisierung aus Haskell-Sicht (2)

...sowie für die darauf aufbauenden Verarbeitungsfunktionen:

Wertpapierdaten:

```
auswertung :: (Float,Float) -> (Float,Float,Float)
reihenausw :: [(Float,Float)] -> [(Float,Float,Float)]
```

Wasserstandsdaten:

```
auswertung' :: (Float,Float) -> (Float,Float,Float)
reihenausw' :: [(Float,Float)] -> [(Float,Float,Float)]
```

Ortsdaten:

```
auswertung'' :: (Float,Float) -> (Float,Float,Float)
reihenausw'' :: [(Float,Float)] -> [(Float,Float,Float)]
```

Konsequenz (1)

Aufgrund der jeweiligen wechselweisen Typgleichheit von

- `auswertung`, `auswertung'`, `auswertung''`
- `reihenausw`, `reihenausw'`, `reihenausw''`

...und der jeweiligen wechselweisen Typgleichheiten von

- `Kursausschlag`, `Messung`, `Ebenenpunkt`
- `Kursverlauf`, `Messreihe`, `Flaeche`

Konsequenz (2)

...arbeiten die Funktionen

- `auswertung`, `auswertung'`, `auswertung''`

typfehlerfrei auf jedem Wert der Typen

- `Kursausschlag`, `Messung`, `Ebenenpunkt`

Das Gleiche gilt für die Funktionen

- `reihenausw`, `reihenausw'`, `reihenausw''`

für jeden Wert der Typen

- `Kursverlauf`, `Messreihe`, `Flaeche`

Daten können so Argument von Funktionen werden, für die das gemäß der mit den Typsynonymen ausgedrückten intendierten Semantik nicht möglich sein sollte.

Typsicherheit sieht anders aus (1)

Die Funktionen zur Wertpapierdatenverarbeitung sind auch auf Wasserstandsdaten und Ortsdaten anwendbar ohne auf Typfehler zu führen:

Wasserstandsdaten

```
auswertung mess1 ->> (5.0,8.0,6.5)
auswertung mess2 ->> (7.2,9.6,8.4)

reihenausw messreihe1
  ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw messreihe2
  ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Ortsdaten

```
auswertung xy1 ->> (5.0,8.0,6.5)
auswertung xy2 ->> (7.2,9.6,8.4)

reihenausw flaeche1
  ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw flaeche2
  ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Typsicherheit sieht anders aus (2)

Die Funktionen zur Wasserstandsdatenverarbeitung sind auch auf Wertpapierdaten und Ortsdaten anwendbar ohne auf Typfehler zu führen:

Wertpapierdaten

```
auswertung' ausschlag1 ->> (5.0,8.0,6.5)
auswertung' ausschlag2 ->> (7.2,9.6,8.4)

reihenausw' verlauf1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw' verlauf2
->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Ortsdaten

```
auswertung' xy1 ->> (5.0,8.0,6.5)
auswertung' xy2 ->> (7.2,9.6,8.4)

reihenausw' flaeche1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw' flaeche2
 ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Typsicherheit sieht anders aus (3)

Die Funktionen zur Ortsdatenverarbeitung sind auch auf Wertpapierdaten und Wasserstandsdaten anwendbar ohne auf Typfehler zu führen:

Wertpapierdaten

```
auswertung'' ausschlag1  ->> (5.0,8.0,6.5)
auswertung'' ausschlag2 ->> (7.2,9.6,8.4)

reihenausw'' verlauf1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw'' verlauf2
->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Wasserstandsdaten

```
auswertung'' mess1 ->> (5.0,8.0,6.5)
auswertung'' mess2 ->> (7.2,9.6,8.4)

reihenausw'' messreihe1
 ->> [(5.0,8.0,6.5),(7.2,9.4,8.3),(1.5,4.1,2.8)]
reihenausw'' messreihe2
 ->> [(2.4,7.8,5.1),(3.2,5.4,4.3),(2.5,8.1,5.3)]
```

Typsicherheit sieht anders aus

Was mit

- Wertpapierdaten, Wasserständen, Ortsdaten

möglich ist, ist auch mit Verarbeitungsfunktionen anderer Typsynonyme möglich, z.B. für

- Meilen und Kilometer, Pferdestärken und Kilowatt, Kraftpfund (engl. pound-force, lb, lbf)) und Newton, etc.

```
type Meile      = Float
type Kilometer = Float
type PS         = Float
...
```

Eine Petitesse? Fragen Sie die NASA!

Marssonde "Climate Orbiter"

Ereignis

- Totalverlust der Sonde Mars Climate Orbiter am 23.09.1998 beim Versuch in die Marsatmosphäre einzutauchen.

Ursache

- Programmierfehler aufgrund widersprüchlicher Verwendung metrischer und nichtmetrischer Daten zwischen Lockheed Martin Astronautics und NASA-Labor Jet Propulsion Laboratory (JPL):

  Pound-force vs. Newton und weiterer metrischer und SI-Einheiten.

Schadenssumme

- Rd. 125 Millionen US$.

Marssondendebakel, Hintergrundmaterial

NASA-Datenbank “Lessons Learned”:

http://www-bcf.usc.edu/∼meshkati/fea03/appendix_html/Mars%20Climate%20Orbiter%20NASA%20LLIS%Database.htm

NASA-Berichte zu Sondenverlust (...0930) und Ursache (...1110):

http://mars.nasa.gov/msp98/news/mco990930.html

http://mars.nasa.gov/msp98/news/mco991110.html

Weitere zusammenfassende Berichte (...0052) und über ignorierte Warnhinweise in der Anflugphase (...report/):

http://www.sciencedirect.com/science/article/pii/S0263786309000052

https://www.wired.com/2010/11/1110mars-climate-observer-report/

Typsynonyme (1)

...verhindern nicht, die sprichwörtlichen Äpfel und Birnen

```
type Apfel = String
type Birne = String

jonathan = "Jonathan" :: Apfel
williams = "Williams" :: Birne

apfel    = "Williams" :: Apfel
birne    = "Jonathan" :: Birne
```

...erfolgreich miteinander zu vergleichen:

```
[apfel,jonathan] == [williams,birne] ->> True
```

Typsynonyme (2)

...erlauben intendierte Typ- und Wertbenutzungen anzuzeigen,

- nicht aber durchzusetzen.

Typsynomyme können deshalb nicht verhindern, Funktionen

- dezidiert zur Verarbeitung von z.B. Wertpapier-, Wasserstands- und Ortsdaten

versehentlich, irrtümlich oder absichtlich auch auf

- ungeeignete und nicht dafür vorgesehene Daten

anzuwenden.

Zurück zu unserem Beispiel

- Zwischen Aktienkursen, Pegelständen und Ortsangaben besteht kein innerer oder bedeutungsmäßiger Zusammenhang.
- Es gibt keinen vernünftigen Grund, Funktionen für Wertpapierdaten auf Wasserstandsdaten und Ortsdaten anzuwenden und umgekehrt.
- Eine gute Programmiersprache sollte erlauben, ungeeignete Verwendung auszuschließen.
  - Programmiersprachliches Mittel: Typsysteme!
  - In Haskell: Neue Typen (und als Verallgemeinerung algebraische Datentypen, siehe Kapitel 5).

Neue Typen, newtype-Deklarationen

Wir ersetzen Typsynynonym-Deklarationen

```
type Kurs       = Float
type Pegelstand = Float
type Koordinate = Float
```

durch newtype-Deklarationen:

$$\texttt{newtype}\ \underbrace{\texttt{Kurs}}_{\textit{Typname}}\quad = \underbrace{\texttt{K}\ \overbrace{\texttt{Float}}^{\textit{Nutzdatentyp}}}_{\textit{Datenkonstruktorname}}$$

```
newtype Pegelstand = Pgl Float
newtype Koordinate = Koordinate Float
```

Beachte: Typname und Datenkonstruktorname dürfen übereinstimmen; sie sind durch den Anwendungskontext unterscheidbar.

Alles andere bleibt (syntaktisch) gleich

...am Beispiel für Wertpapierdaten:

```
newtype Kurs              = K Float
type Niedrigst            = Kurs
type Hoechst              = Kurs
type Geglaettet           = Kurs
type Kursausschlag        = (Niedrigst,Hoechst)
type Ausschlagsanalyse    = (Niedrigst,Hoechst,Geglaettet)
type Kursverlauf          = [Kursausschlag]
type Verlaufsanalyse      = [Ausschlagsanalyse]
```

Beachte allerdings:

- Wie Kurs sind auch Niedrigst, Hoechst und Geglaettet keine Synonyme mehr für `Float`, sondern für den neuen Typ Kurs.
- In gleicher Weise sind auch Kursausschlag, Ausschlagsanalyse, Kursverlauf und Verlaufsanalyse keine Synonyme mehr für `Float`-Paare, `Float`-Tripel und Listen von `Float`-Paaren und `Float`-Tripeln.

Anpassung der Funktionsdefinitionen

..Konstruktoren in Argumentmustern, alles andere (syntaktisch) gleich:

```
auswertung :: Kursausschlag -> Ausschlagsanalyse
 auswertung  (K x,K y) = (K x,K y,K arithMittel)
  where arithMittel = (x+y) / 2

 reihenausw :: Kursverlauf -> Verlaufsanalyse
 reihenausw [] = []
 reihenausw ((K x,K y)) : xys)
               = (auswertung (K x, K y)) : reihenausw xys
```

statt (wie mit Typsynonymen in Kapitel 4.1):

```
 auswertung :: Kursausschlag -> Ausschlagsanalyse
 auswertung (x,y) = (x,y,arithMittel)
  where arithMittel = (x+y) / 2

 reihenausw :: Kursverlauf -> Verlaufsanalyse
 reihenausw [] = []
 reihenausw ((x,y) : xys)
               = (auswertung (x,y)) : reihenausw xys
```

Analog für Wasserstandsdaten

Modifizierte Typdeklarationen:

```
newtype Pegelstand       = Pgl Float
type Niedrig             = Pegelstand
type Hoch                = Pegelstand
type Mittel              = Pegelstand
type Messung             = (Niedrig,Hoch)
type Auswertung          = (Niedrig,Hoch,Mittel)
type Messreihe           = [Messung]
type Auswertungsreihe = [Auswertung]
```

Angepasste Funktionsdefinitionen:

```
auswertung' :: Messung -> Auswertung
auswertung' (Pgl x,Pgl y) = (Pgl x,Pgl y,Pgl arithMittel)
 where arithMittel = (x+y) / 2

reihenausw' :: Messreihe -> Auswertungsreihe
reihenausw' [] = []
reihenausw' ((Pgl x,Pgl y) : xys)
              = (auswertung' (Pgl x,Pgl y)) : reihenausw' xys
```

Analog für Ortsdaten

Modifizierte Typdeklarationen:

```
newtype Koordinate = Koordinate Float
type X            = Koordinate
type Y            = Koordinate
type Z            = Koordinate
type Ebenenpunkt  = (X,Y)
type Raumpunkt    = (X,Y,Z)
type Flaeche      = [Ebenenpunkt]
type Koerper      = [Raumpunkt]
```

Angepasste Funktionsdefinitionen:

```
auswertung'' :: Ebenenpunkt -> Raumpunkt
auswertung'' (Koordinate x,Koordinate y)
  = (Koordinate x,Koordinate y,Koordinate arithMittel)
    where arithMittel = (x+y) / 2

reihenausw'' :: Flaeche -> Koerper
reihenausw'' [] = []
reihenausw'' ((Koordinate x,Koordinate y) : xys)
 = (auswertung'' (Koordinate x,Koordinate y)) : reihenausw'' xys
```

Typsicherheit erreicht!

...die (Nutz-) Daten, die Gleitkommawerte, liegen jetzt geschützt hinter Datenkonstruktoren:

Wertpapierdaten:

```
ausschlag = (K 5.0,K 8.0)
verlauf   = [(K 5.0,K 8.0),(K 7.2,K 9.4),(K 1.5,K 4.1)]
```

Wasserstandsdaten:

```
messung   = (Pgl 5.0,Pgl 8.0)
messreihe = [(Pgl 5.0,Pgl 8.0),(Pgl 7.2,Pgl 9.4),
             (Pgl 1.5,Pgl 4.1)]
```

Ortsdaten:

```
Ebenenpunkt = (Koordinate 5.0,Koordinate 8.0)
flaeche     = [(Koordinate 5.0,Koordinate 8.0),
               (Koordinate 7.2,Koordinate 9.4),
               (Koordinate 1.5,Koordinate 4.1)]
```

Wertpapierdaten typgesichert typsicher!

Wertpapierdatenverarbeitungsfunktionen:

```
auswertung :: Kursausschlag  -> Ausschlagsanalyse
reihenausw :: Kursverlauf -> Verlaufsanalyse
```

Anwendbarkeit auf Wertpapierdaten wie gewünscht:

```
auswertung ausschlag ->>  (K 5.0,K 8.0,K 6.5)
reihenausw verlauf   ->> [(K 5.0,K 8.0,K 6.5),
                          (K 7.2,K 9.4,K 8.3),
                          (K 1.5,K 4.1,K 2.8)]
```

Keine Anwendbarkeit auf Wasserstands-, Ortsdaten oder Gleitkommaz.:

```
auswertung messung     ->> "Fehler: Typen passen nicht."
reihenausw messreihe   ->> "Fehler: Typen passen nicht."

auswertung ebenenpunkt ->> "Fehler: Typen passen nicht."
reihenausw flaeche     ->> "Fehler: Typen passen nicht."

auswertung (5.0,8.0)   ->> "Fehler: Typen passen nicht."
reihenausw [5.0,8.0),(7.2,9.4),(1.5,4.1)]
                       ->> "Fehler: Typen passen nicht."
```

In gleicher Weise auch

...Wasserstands- und Ortsdaten jetzt typgesichert typsicher!

Wasserstandsdaten:

```
auswertung' :: Messung   ->  Auswertung
reihenausw' :: Messreihe -> Auswertungsreihe
```

Ortsdaten:

```
auswertung'' :: Ebenenpunkt  ->  Raumpunkt
reihenausw'' :: Flaeche -> Koerper
```

Mission erfüllt: Typsicherheit erreicht!

Das Konzept neuer Typen, Nutzdaten hinter Datenkonstruktoren zu verbergen, erlaubt

- zusätzlich zum Anzeigen intendierter Typ- und Wertbenutzungen, diese auch durchzusetzen

und (Nutz-) Daten so vor

- versehentlicher wie bewusster Verarbeitung durch nicht dafür vorgesehene Funktionen effektiv zu schützen.

Kapitel 4.3

Typklassen

Typsicherheit erreicht. Mission erfüllt? (1)

Betrachte folgende Wertpapierdaten:

- Typgesichert (`newtype Kurs = K Float`):

```
gs_kurs1       =  K 5.0
gs_kurs2       =  K 7.2
gs_ausschlag1 = (K 5.0,K 8.0)
gs_ausschlag2 = (K 7.2,K 9.6)
gs_verlauf1    = [(K 5.0,K 8.0),(K 7.2,K 9.4),
                   (K 1.5,K 4.1)]
gs_verlauf2    = [(K 2.4,K 7.8),(K 3.2,K 5.4),
                   (K 2.5,K 8.1)]
```

- Typungesichert (`type Kurs = Float`):

```
ugs_kurs1       = 5.0
ugs_kurs2       = 7.2
ugs_ausschlag1 = (5.0,8.0)
ugs_ausschlag2 = (7.2,9.6)
ugs_verlauf1    = [(5.0,8.0),(7.2,9.4),(1.5,4.1)]
ugs_verlauf2    = [(2.4,7.8),(3.2,5.4),(2.5,8.1)]
```

Typsicherheit erreicht. Mission erfüllt? (2)

...und die Ergebnisse folgender Ausdrucksauswertungen:

```
ugs_kurs1 == ugs_kurs1 ->> True
ugs_kurs1 /= ugs_kurs2 ->> True

gs_kurs1  == gs_kurs1  ->> "Fehler: (==) unbekannt"
gs_kurs1  /= gs_kurs2  ->> "Fehler: (/=) unbekannt"

ugs_ausschlag1 == ugs_ausschlag1 ->> True
ugs_ausschlag1 /= ugs_ausschlag2 ->> True

gs_ausschlag1  == gs_ausschlag1  ->> "Fehler: (==) unbekannt"
gs_ausschlag1  /= gs_ausschlag2  ->> "Fehler: (/=) unbekannt"

ugs_verlauf1 == ugs_verlauf1 ->> True
ugs_verlauf1 /= ugs_verlauf2 ->> True

gs_verlauf1  == gs_verlauf1  ->> "Fehler: (==) unbekannt"
gs_verlauf1  /= gs_verlauf2  ->> "Fehler: (/=) unbekannt"
```

Schutz vor missbräuchlicher Nutzung überschießend? Auch intendierte Zugriffe und Verarbeitung nicht mehr möglich?

Ad hoc Abhilfe (1)

Definiere Gleichheitstests für Kurse, Kursausschläge, Kursverläufe:

```
k_eq :: Kurs -> Kurs -> Bool
(K k1) 'k_eq' (K k2) = k1==k2

k_neq :: Kurs -> Kurs -> Bool
k1 'k_neq' k2 = not (k1 'k_eq' k2)

ka_eq :: Kursausschlag -> Kursausschlag -> Bool
(K k1,K k2) 'ka_eq' (K k3,K k4) = (k1,k2)==(k3,k4)

ka_neq :: Kursausschlag -> Kursausschlag -> Bool
ka1 'ka_neq' ka2 = not (ka1 'ka_eq' ka2)

kv_eq :: Kursverlauf -> Kursverlauf -> Bool
[] 'kv_eq' [] = True
(ka:kas) 'kve_q' (la:las) = ka 'ka_eq' la && kas 'kv_eq' las
_ 'kv_eq' _ = False

kv_neq :: Kursverlauf -> Kursverlauf -> Bool
kv1 'kv_neq' kv2 = not (kv1 'kv_eq' kv2)
```

Ad hoc Abhilfe (2)

Wir erhalten folgende Auswertungsergebnisse:

```
gs_kurs1      'k_eq'   gs_kurs1      ->> True
gs_kurs1      'k_neq'  gs_kurs2      ->> True
gs_ausschlag1 'ka_eq'  gs_ausschlag1 ->> True
gs_ausschlag1 'ka_neq' gs_ausschlag2 ->> True
gs_verlauf1   'kv_eq'  gs_verlauf1   ->> True
gs_verlauf1   'kv_neq' gs_verlauf2   ->> True
```

Ad hoc Abhilfe erfüllt den Zweck.

Ad hoc Abhilfe generalisierbar?

Gleichheits- und Ungleichheitstests

- wären jetzt in gleicher Weise für Wasserstands- und Ortsdaten zu definieren.
- Aufwändig, unpraktisch, wenig elegant; allein die Wahl der vielen Namen ist bereits in hohem Maß unschön.

Haskell bietet ein zweckmäßigeres und schlagkräftigeres Sprachmittel an:

- Typklassen

Typklassen in Haskell

Typklassen

- haben Typen als Elemente.
- legen eine Menge von Operationen und Relationen fest, die auf ihren Elementen implementiert sein müssen.
- können für diese Operationen und Relationen bereits vollständige Standardimplementierungen oder noch zu vervollständigende Protoimplementierungen vorsehen.

Typen

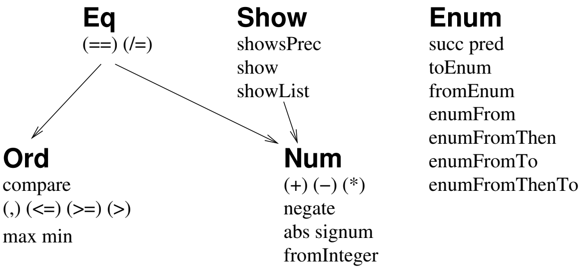
- werden durch Instanzbildung zu Elementen bzw. Instanzen einer Typklasse.

Beispiele vordefinierter Typklassen

- Typklasse `Eq`: Werte von Typen dieser Typklasse müssen auf Gleichheit und Ungleichheit vergleichbar sein.
- Typklasse `Ord`: Werte von Typen dieser Typklasse müssen über Gleichheit und Ungleichheit hinaus bezüglich ihrer relativen Größe vergleichbar sein.
- Typklasse `Num`: Werte von Typen dieser Typklasse müssen mit ausgewählten numerischen Operationen verknüpfbar sein, d.h. addiert, multipliziert, subtrahiert, etc. werden können.
- Typklasse `Show`: Werte von Typen dieser Typklasse müssen eine Darstellung in Form von Zeichenreihen haben.
- Typklasse `Enum`: Werte von Typen dieser Typklasse müssen aufzählbar sein.

Typklassen

...bilden eine Hierarchie:



Quelle: Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach. Addison-Wesley, 1999, Abb. 2.4 (Ausschnitt).

Typklasse Eq

...für Typen, deren Werte absolut verglichen werden können, d.h. auf Gleichheit, Ungleichheit:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)
```

Die Typklasse Eq

- verlangt von Instanzen die Implementierung von zwei Wahrheitswertfunktionen (oder Prädikaten): (==), (/=).
- stellt für beide Wahrheitswertfunktion eine Protoimplementierung zur Verfügung.

Minimalvervollständigung bei Instanzbildungen für Eq:

- Implementierung von entweder (==) oder (/=).

Bemerkung

Die Protoimplementierungen für sich allein sind

- unvollständig und nicht ausreichend, da sie sich wechselweise aufeinander abstützen.

Dennoch ergibt sich folgender Vorteil aus ihrer Angabe:

- Bei Instanzbildungen reicht es, entweder eine Implementierung für `(==)` oder für `(/=)` anzugeben. Für den jeweils anderen Operator ist dann die Protoimplementierung vollständig.
- Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Protoimplementierungen überschrieben.

Instanzbildung für die Typklasse Eq (1)

...am Beispiel des Typs Bool der Wahrheitswerte:

```
instance Eq Bool where
 True  == True  = True
 False == False = True
     _ == _     = False
```

Alternativ und gleichwertig:

```
instance Eq Bool where
 True  /= True  = False
 False /= False = False
     _ /= _     = True
```

Instanzbildung für die Typklasse Eq (2)

Am Beispiel eines Typs Punkt für Punkte in der ganzzahligen (x,y)-Ebene:

```
newtype Punkt = Pkt (Int,Int)

instance Eq Punkt where
  (Pkt (x,y)) == (Pkt (u,v)) = (x==u) && (y==v)
```

Typklasse `Ord` (1)

...für Typen, deren Werte relativ verglichen werden können:

```
class Eq a => Ord a where
 compare              :: a -> a -> Ordering
 (<), (<=), (>), (>=) :: a -> a -> Bool
 max, min             :: a -> a -> a
 compare x y
  | x == y    = EQ
  | x <= y    = LT
  | otherwise = GT
 x <= y       = compare x y /= GT
 x < y        = compare x y == LT
 x >= y       = compare x y /= LT
 x > y        = compare x y == GT
 max x y
  | x <= y    = y
  | otherwise = x
 min x y
  | x <= y    = x
  | otherwise = y
```

Typklasse `Ord` (2)

Die Typklasse `Ord`

- verlangt von Instanzen die Implementierung der Funktionen `compare`, `max` und `min` sowie der Prädikate `(<)`, `(<=)`, `(>=)`, `(>)`.
- stellt für alle dieser Funktionen und Prädikate Protoimplementierungen zur Verfügung.

Minimalvervollständigung bei Instanzbildungen für `Ord`:

- Implementierung von entweder `compare` oder `(<=)`.

Typklasse Show

...für Typen, deren Werte als Zeichenreihe dargestellt werden können:

```
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS

  showsPrec _ x s = show x ++ s
  show x = showsPrec 0 x ""
  showList [] = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
                    where showl []     =  showChar ']'
                          showl (x:xs) =
                           showChar ',' . shows x . showl xs
```

Minimalvervollständigung bei Instanzbildungen für Show:

- Implementierung von entweder show oder showPrec.

Typklasse Read

...für Typen, deren Werte aus einer Zeichenreihe abgeleitet werden können:

```
type ReadS a = String -> [(a,String)]

class Read a where
 readsPrec :: Int -> ReadS a
 readList  :: ReadS [a]
 readList = ...
```

Minimalvervollständigung bei Instanzbildungen für `Read`:

- Implementierung von `readsPrec`.

...siehe Standard-Präludium und Sprachbericht für hier nicht angegebene Hilfsfunktionen von `Show` und `Read`.

Typklasse Enum

...für Typen, deren Werte aufgezählt werden können:

```
class Enum a where
 succ, pred     :: a -> a
 toEnum         :: Int -> a
 fromEnum       :: a -> Int
 enumFrom       :: a -> [a]                 -- [n..]
 enumFromThen   :: a -> a -> [a]            -- [n,n'..]
 enumFromTo     :: a -> a -> [a]            -- [n..m]
 enumFromThenTo :: a -> a -> a -> [a]       -- [n,n'..m]
 succ             = toEnum . (+1) . fromEnum
 pred             = toEnum . (subtract 1) . fromEnum
 enumFrom x       = map toEnum [fromEnum x ..]
 enumFromTo x     = map toEnum [fromEnum x .. fromEnum y]
 enumFromThen x y =
  map toEnum [fromEnum x, fromEnum y .. fromEnum]
 enumFromThenTo x y z =
  map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Minimalvervollständigung bei Instanzbildungen für Enum:

- Implementierung von toEnum und fromEnum.

Typklasse Num

...für Typen, deren Werte numerisch behandelt werden können:

```
class (Eq a, Show a) => Num a where
 (+), (-), (*) :: a -> a -> a
 negate :: a -> a
 abs, signum :: a -> a
 fromInteger :: Integer -> a
 x - y    = x + negate y
 negate x = 0 - x
```

Minimalvervollständigung bei Instanzbildungen für Num:

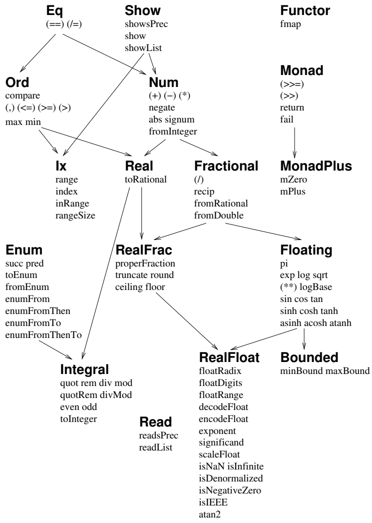
- Implementierung aller Funktionen mit Ausnahme von entweder negate oder (-).

Weitere numerische Typklassen neben Num

```
class (Num a, Ord a)    => Real a where...
class (Real a, Enum a) => Integral a where...
class (Num a)          => Fractional a where...
class (Fractional a)           => Floating a where...
class (Real a, Fractional a)   => RealFrac a where...
class (RealFrac a, Floating a) => RealFloat a where...
```

...siehe Standard-Präludium und Sprachbericht für Einzelheiten.

Die Typklassenhierachie im Überblick



Quelle: Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach. Addison-Wesley, 1999, Abb. 2.4.

Beschreibung einiger Typklassen

- Typklasse 'Gleichheit' `Eq`: die Klasse der Typen mit Gleichheits- (`==`) und Ungleichheitsrelation (`/=`).
- Typklasse 'Ordnung' `Ord`: die Klasse der Typen mit Ordnungsrelationen ($<$, $\leq$, $>$, $\geq$, etc.).
- Typklasse 'Numerisch' `Num`: die Klasse der Typen, deren Werte sich numerisch verhalten (Bsp.: `Int`, `Integer`, `Float`, `Double`)
- Typklasse 'Aufzählung' `Enum`: die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29] :: Int`).
- Typklasse 'Ausgabe' `Show`: die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können.
- Typklasse 'Eingabe' `Read`: die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind.
- ...

Haskells Philosophie zu Typen und Typklassen

Faustregel: Bei Einführung eines neuen Typs

- überlege, welche Operationen und Relationen auf Werte dieses Typs anwendbar sein sollen und auf die Werte welcher bereits eingeführter Typen diese oder vergleichbare Operationen und Relationen ebenfalls anwendbar sind,
- mache den neuen Typ zu einer Instanz all derjenigen Typklassen, zu denen diese anderen Typen gehören; oft reicht dafür eine `deriving`-Klausel aus.

- Sind auf die Werte des neuen Typs Operationen und Relationen anwendbar, die noch nicht in dieser oder vergleichbarer Form in einer Typklasse gebündelt sind, so
    - führe eine neue Typklasse mit diesen Funktionen und Relationen ein; ggf. zusammen mit vollständigen Implementierungen oder zu vervollständigenden Protoimplementierungen, wo passend.

Instanzbildungen für die Typklasse Eq (1)

Instanzbildung: Mache Typ Kurs zu einem Element von Eq:

```
newtype Kurs = K Float

instance Eq Kurs where
 K k1 == K k2 = k1==k2
```

Mit dieser Instanzbildung jetzt sofort möglich:

```
gs_kurs1      == gs_kurs1      ->> True
gs_kurs1      /= gs_kurs2      ->> True

gs_ausschlag1 == gs_ausschlag1 ->> True
gs_ausschlag1 /= gs_ausschlag2 ->> True

gs_verlauf1   == gs_verlauf1   ->> True
gs_verlauf1   /= gs_verlauf2   ->> True
```

Instanzbildungen für die Typklasse Eq (2)

Analog die Instanzbildung für `Pegelstand` und Koordinate:

```
newtype Pegelstand = Pgl Float
instance Eq Pegelstand where
 Pgl p1 == Pgl p2 = p1==p2
```

Tatsächlich ist sogar eine automatische Instanzbildung möglich:

```
newtype Koordinate = Koordinate Float deriving Eq
```

Die Typdeklaration mit `deriving`-Klausel hat dieselbe Bedeutung wie die Instanzbildung:

```
newtype Koordinate = Koordinate Float
instance Eq Koordinate where
 Koordinate k1 == Koordinate k2 = k1==k2
```

Instanzbildungen für die Typklasse Ord

```
newtype Kurs = K Float
instance Ord Kurs where
  K k1 <= K k2 = k1 <= k2
```

Mit dieser Instanzbildung jetzt sofort möglich:

```
gs_kurs1 <= gs_kurs1 ->> False
gs_kurs1 > gs_kurs2 ->> False

gs_ausschlag1 >= gs_ausschlag1 ->> False
gs_ausschlag1 'max' gs_ausschlag2 ->> True

gs_verlauf1 'min' gs_verlauf1 ->> True
gs_verlauf1 'compare' gs_verlauf2 ->> GT
```

Analog die Instanzbildung für `Pegelstand` und `Koordinate`.

Ausschließlich automatische Instanzbildungen

...sind für unser Beispiel am bequemsten:

```
newtype Kurs        = K Float
                          deriving (Eq,Ord,Show)
newtype Pegelstand = Pgl Float
                          deriving (Eq,Ord,Show)
newtype Koordinate = Koordinate Float
                          deriving (Eq,Ord,Show)
```

Selbstdefinierte Typklassen, Wiederverwendung

...auf Wertpapier-, Wasserstands- und Ortsdaten sind Operationen zur Auswertung einzelner Ereignisse und Ereignisfolgen anwendbar.

Alle diese Funktionen haben ähnliche Funktionalität und ähnliche Namen:

- `auswertung`, `auswertung'`, `auswertung''`
- `reihenausw`, `reihenausw'`, `reihenausw''`

Es bietet sich deshalb an,

- diese Operationen in einer selbstdefinierten neuen Typklasse zu bündeln,
- so Namen einzusparen und wiederzuverwenden,
- die Verwandtheit der Typen für Wertpapier-, Wasserstands- und Ortsdaten auszudrücken.

Analysierbar: 1. selbstdefinierte Typklasse

Wir bündeln die Auswertungsfunktionen auf Wertpapier-, Wasserstands- und Ortsdaten in der selbstdefinierten neuen Typklasse Analysierbar:

```
class (Ord a, Fractional a) => Analysierbar a where
 auswertung  :: (a,a) -> (a,a,a)
 reihenausw  :: [(a,a)] -> [(a,a,a)]
 arithMittel :: a -> a -> a

 arithMittel x y = (x+y) / 2          -- Protoimple-
 reihenausw [] = []                   -- mentierungen
 reihenausw ((x,y) : xys)
              = (auswertung (x,y)) : reihenausw xys
```

Typklasse Analysierbar: Vorbereitungen

...zur Instanzbildung für Kurs, Pegelstand und Koordinate.

```
newtype Kurs       = K Float
                             deriving (Eq,Ord,Show)
newtype Pegelstand = Pgl Float
                             deriving (Eq,Ord,Show)
newtype Koordinate = Koordinate Float
                             deriving (Eq,Ord,Show)
```

Nötige, aber nicht automatisch ableitbare Instanzbildungen:

```
instance Num Kurs where...

instance Num Pegelstand where...

instance Num Koordinate where...

instance Analysierbar Float where
 auswertung (f1,f2) = (f1,f2,arithMittel f1 f2)
```

Typklasse Analysierbar: Instanzbildungen

...für Kurs, Pegelstand und Koordinate.

```
instance Analysierbar Kurs where
 auswertung (K k1,K k2)
  = (K k1,K k2,K (arithMittel k1 k2))

instance Analysierbar Pegelstand where
 auswertung (Pgl p1,Pgl p2)
  = (Pgl p1,Pgl p2,Pgl (arithMittel p1 p2))

instance Analysierbar Koordinate where
 auswertung (Koordinate k1,Koordinate k2)
  = (Koordinate k1,Koordinate k2,
     Koordinate (arithMittel k1 k2))
```

Warnung: 2. selbstdefinierte Typklasse

...die Auswertung von Wertpapier- und Wasserstandsdaten mag vorteilhafte oder gefährliche Situationen erkennen lassen, die entsprechende 'Warnungen' ermöglichen sollte.

Es liegt nahe, diese Funktionen in einer weiteren neuen Typklasse Warnung zu bündeln:

```
class (Analysierbar a) => Warnung a where
 warnung :: (a,a) -> String
 warnreihe :: [(a,a)] -> String
 warnreihe xys = warnung (wr xys (0,0)) -- Proto-
  where wr [] pq = pq                   -- implementierung
        wr ((x,y) : xys) (p,q) = wr xys (x+p,y+q)
```

Typklasse Warnung: Instanzbildungen (1)

```
instance Warnung Kurs where
 warnung (K k1,K k2)
  | k2 > 9*k1 = "Verkaufen! Aktie zu spekulativ."
  | k2 > 6*k1 = "Halten! Aktie an Spekulationsschwelle."
  | k2 > 3*k1 = "Zukaufen! Aktie hat Phantasie."
  | otherwise = "Verkaufen! Aktie ohne Phantasie."

 -- Für warnreihe passt die Standardimplementierung.
 -- Nichts zu tun.
```

Typklasse Warnung: Instanzbildungen (2)

```
instance Warnung Pegelstand where
 warnung (Pgl p1,Pgl p2)
  | p2 >= 100 = "Evakuieren! Deich kurz vor Bruch."
  | p2 >=  80 = "Achtung! Deich an Belastungsgrenze."
  | p1 <=  20 = "Sperrwerk öffnen! Pegel zu niedrig."
  | otherwise = "Pegel im Normalbereich."

 -- Für warnreihe passt die Standardimplementierung nicht.
 -- Wir überschreiben sie daher:
 warnreihe pgs = meldung anteil
  where
   anzahlEreignisse = length pgs
   anzahlGefahrEreignisse
    = length [max | (Pgl min,Pgl max) <- pgs, max >= 100]
   anteil = (anzahlGefahrEreignisse * 100)
                `div` anzahlEreignisse
   meldung n
    | n > 30    = "Anteil Gefahrereignisse hoch"
    | n > 10    = "Anteil Gefahrereignisse moderat"
    | otherwise = "Anteil Gefahrereignisse gering"
```

Typklasse Warnung: Instanzbildungen (3)

Für Ortsdaten als Punkte im zwei- bzw. drei- dimensionalen mathematischen Raum besteht

- kein Grund, Warnungen auszugeben.

Deshalb ist eine Instanzbildung von `Koordinate` für die Typklasse `Warnung` unnötig und wird

- unterlassen.

Vordefinierte Instanzen von Typklassen

Viele Typen, inbesondere

- Elementartypen (`Bool`, `Char`, `Int`,...)
- Tupel von Elementartypen
- Listen von Elementartypen (speziell Zeichenreihen)
- ...

sind bereits vordefinierte Instanzen der passenden Typklassen.

Deshalb sind wir damit ausgekommen, Instanzbildungen für

- `Kurs`, `Pegelstand` und `Koordinate` vorzunehmen.

Auf Tupel- und Listentypen wie `Kursausschlag`, `Kursverlauf`, etc., haben sich die Eigenschaften automatisch übertragen.

Typsicherheit bleibt gewährt!

Beachte:

- Die Funktionen `auswertung`, `reihenausw` sind auf
  - Wertpapier-, Wasserstands- und Ortsdaten anwendbar.
- Die Funktionen `warnung`, `warnreihe` sind auf
  - Wertpapier- und Wasserstandsdaten anwendbar.

Dennoch: Typsicherheit wird nicht korrumpiert und bleibt gewahrt:

- Alle Aufrufe erfolgen mit typspezifischem Code!

Die Typsicherheit bleibt deshalb in vollem Umfang gewährt.

Typklassen vs. objektorientierter Klassen

Typklassen in Haskell unterscheiden sich wesentlich vom Klassenkonzept objektorientierter Sprachen.

Objektorientiert: Klassen

- dienen der Strukturierung von Programmen.
- liefern Blaupausen zur Generierung von Werten.

In Haskell: Typklassen

- sind Sammlungen von Typen, deren Werte in 'ähnlicher' Weise verarbeitet werden können.
- erhalten Typen als Elemente explizit durch Instanzbildung oder implizit durch automatische Instanzbildung (`deriving`-Klausel) zugewiesen.
- dienen nicht der Strukturierung von Programmen; liefern keine Blaupausen zur Generierung von Werten.

Übungsaufgabe 4.3.1

Vergleiche das Klassenkonzept aus Haskell mit dem Schnittstellenkonzept aus Java.

Welche Gemeinsamkeiten, welche Unterschiede gibt es?

Ausblick

...im Zusammenhang mit

- Datentypdeklarationen (Kapitel 5)
- *Ad hoc* Polymorphie, Überladung (Kapitel 11)

kommen wir auf

- Typdefinitionen
- Typklassen

noch einmal zurück.

Kapitel 4.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 4 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7.1, Typsynonyme mit type; Kapitel 7.2, Einfache algebraische Typen mit data und newtype; Kapitel 7.4, Automatische Instanzen von Typklassen; Kapitel 7.8, Eigene Klassen definieren)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 2, Expressions, types and values; Kapitel 3, Numbers)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 4 (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 2, Believe the Type)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 3, Defining Types, Streamlining Functions; Kapitel 6, Using Typeclasses)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13.4, A tour of the built-in Haskell classes)

Kapitel 5

Datentypdeklarationen

Grundlegende Datentypstrukturen

...in Programmiersprachen sind:

- Aufzählungstypen
- Produkttypen
- Summentypen

Charakterisierung und typische Beispiele

Aufzählungstypen

- Typen mit jeweils endlich vielen Werten.

  Beispiel: Typ `Jahreszeiten` mit Werten `Fruehling`, `Sommer`, `Herbst` und `Winter`.

Produkttypen (oder Verbundtypen (engl. record types))

- Typen mit möglicherweise unendlich vielen (Tupel-) Werten.

  Beispiel: Typ `Mensch` mit Werten `(Adam,Riese,männlich)`, `(Ada,Lovelace,weiblich)`, etc.

Summentypen (oder Vereinigungstypen)

- Typen mit Werten, die sich aus der Vereinigung der Werte verschiedener Typen mit jeweils möglicherweise unendlich vielen Werten ergeben.

  Beispiel: Typ `Sammelsurium` als Vereinigung der (Werte der) Typen `Buch`, `KFZ`, `Haustier`, etc.

Datentypdeklarationen und Sprachkonstrukte

...in Haskell: type, newtype, data.

Bereits besprochen: Typsynonyme, neue Typen (s. Kapitel 4):

- type-Deklarationen zur Definition von Typsynonymen, d.h. neue Namen für existierende Typen, Typaliase:

```
type Kurs = Float
type Niedrigst = Kurs
type Hoechst = Kurs
type Kursauschlag
 = (Niedrigst,Hoechst)
```

```
type Pegelstand = Float
type Niedrig = Pegelstand
type Hoch = Pegelstand
type Messung
 = (Niedrig,Hoch)
```

```
type Koordinate = Float
type X = Koordinate
type Y = Koordinate
type Ebenenpunkt
 = (X,Y)
```

  ...keine zusätzliche Typsicherheit; unterstützen Transparenz.

- newtype-Deklarationen zur Definition von Typidentitäten:

```
newtype Kurs = K Float
type Niedrigst = Kurs
type Hoechst = Kurs
type Kursauschlag
 = (Niedrigst,Hoechst)
```

```
newtype Pegelstand = Pgl Float
type Niedrig = Pegelstand
type Hoch = Pegelstand
type Messung
 = (Niedrig,Hoch)
```

```
newtype Koordinate = Koordinate Float
type X = Koordinate
type Y = Koordinate
type Ebenenpunkt
 = (X,Y)
```

  ...Typsicherheit durch (Datenwert-) Konstruktoren.

  Allerdings: Beschränkt auf 1 Konstruktor mit 1 (Daten-) Feld.

Neu in diesem Kapitel: data-Deklarationen

Algebraische Datentypen

- data-Deklarationen zur Definition originär neuer Typen, von Aufzählungstypen, Produkttypen und Summentypen.
  - Aufzählungstypen

    ```
    data Jahreszeiten = Fruehling | Sommer
                        | Herbst | Winter
    data Geschlecht   = Maennlich | Weiblich
    ```
  - Produkttypen

    ```
    type Vorname  = String
    type Nachname = String
    data Mensch   = M Vorname Nachname Geschlecht
    ```
  - Summentypen

    ```
    data Baum = Blatt Int | Wurzel Baum Int Baum
    ```

Kapitel 5.1

Algebraische Datentypen

Kapitel 5.1.1

Aufzählungstypen

Beispiele vordefinierter Aufzählungstypen

Typ der Ordnungswerte, 3 Werte:

```
data Ordering = LT | EQ | GT deriving (Eq,Ord,
                                        Bounded,
                                        Enum, Read,
                                        Show)
```

Typ der Wahrheitswerte, 2 Werte:

```
data Bool = False | True deriving (Eq,Ord,Bounded,
                                    Enum,Read,Show)
```

Trivialer Typ (oder Nulltupeltyp), 1 Wert:

```
data () = () deriving (Eq,Ord,Bounded,Enum,Read,
                        Show)
```

...Nulltupeltyp und einziger (def.) Wert ident bezeichnet: ().

Beispiele selbstdefinierter Aufzählungstypen

```
data Jahreszeiten = Fruehling | Sommer
                    | Herbst | Winter
                    deriving (Eq,Ord,Bounded,
                              Enum,Read,Show)

data Spielfarbe   = Karo | Herz | Pik | Kreuz
                    deriving (Eq,Ord,Bounded,
                             Enum,Read,Show)

data Werktage     = Montag | Dienstag | Mittwoch
                    | Donnerstag | Freitag
                    deriving (Eq,Ord,Bounded,
                              Enum,Read,Show)

data Wochenende   = Samstag | Sonntag
                    deriving (Eq,Ord,Bounded,
                              Enum,Read,Show)
```

Kapitel 5.1.2

Produkttypen

Beispiele selbstdefinierter Produkttypen

```
type Vorname    = String               -- Personendaten
type Nachname   = String
data Geschlecht = Maennlich
                  | Weiblich deriving (Eq,Show)
type Gemeinde   = String                 -- Adressdaten
type Strasse    = String
type Hausnr     = Int
type Land       = String

data Person     = P Vorname Nachname Geschlecht d...
data Anschrift  = A Gemeinde Strasse Hausnr Land d...
data Einwohner  = E Land Gemeinde [Person] deriving...
data Wohnsitze  = W Land (Person -> [Anschrift])
data Gemeldet   = G (Land -> Gemeinde -> Strasse
                         -> Hausnr -> [Person])
```

Kapitel 5.1.3

Summentypen

Beispiele vordefinierter Summentypen

Listen

```
data [a] = []
         | a : [a] deriving (Eq,Ord)
                    -- Kein gültiges Haskell;
                     -- nur zur Illustration!
```

Der Möglicherweise-Typ

```
data Maybe a = Nothing
             | Just a deriving (Eq,Ord,Read,Show)
```

Der Entweder/Oder-Typ

```
data Either a b = Left a
                | Right b deriving (Eq,Ord,Read,
                                     Show)
```

Beispiele selbstdefinierter Summentypen (1)

```
type Autor            = String          -- Buch-/E-Buchdaten
type Titel            = String
type Verlag           = String
type Auflage          = Int
type Lieferbar        = Bool
type LizenzBisJahr    = Int
type Hauptdarsteller = [String]                 -- Filmdaten
type Regisseur        = String
type Sprachen         = [String]
type Kuenstler        = String         -- Musikaufnahmedaten
type Std              = Int
type Min              = Int
type Sek              = Int
type Spieldauer       = (Std,Min,Sek)

data BildSchriftUndTontraeger =
      Buch Autor Titel Verlag Auflage Lieferbar
      | E_Buch Autor Titel Verlag LizenzBisJahr
      | DVD Titel Hauptdarsteller Regisseur Sprachen
      | CD Kuenstler Titel Spieldauer deriving (Eq,Show)
```

Beispiele selbstdefinierter Summentypen (2)

```
type Autor         = String              -- Buch-/E-Buchdaten
type Titel         = String
type Verlag        = String
type Auflage       = Int
type Lieferbar     = Bool
data Kategorie     = PKW | LKW | Bus | Cabrio | SUV
                     deriving (Eq,Show)    -- Fahrzeugdaten
type Marke         = String
type Listenpreis   = Float
data Tierart       = Hund | Katze | Maus | Kanarienvogel
                     deriving (Eq,Show)    -- Haustierdaten
type Rufname       = String
type Gewicht_in_kg = Float
type Vielfrass     = Bool

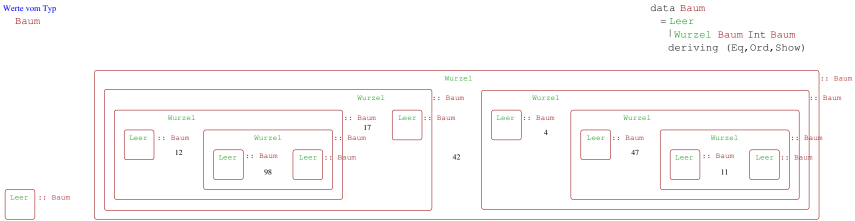
data Sammelsurium =
      Buch Autor Titel Verlag Auflage Lieferbar
      | KFZ Kategorie Marke Listenpreis
      | Haustier Tierart Rufname Gewicht_in_kg Vielfrass
      deriving (Eq,Show)
```

Beispiele rekursiver Summentypen (1)

Zweistellige Bäume, Binärbäume

```
data Baum = Leer
          | Wurzel Baum Int Baum
          deriving (Eq,Ord,Show)
```

Veranschaulichung: Werte vom Typ Baum:

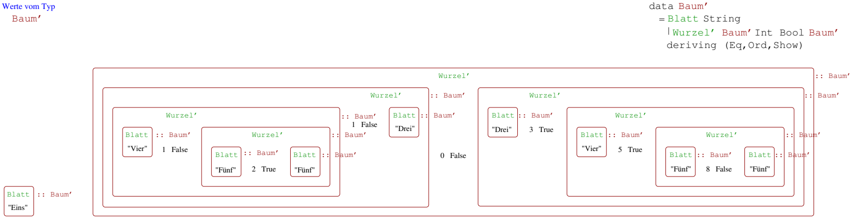


Beispiele rekursiver Summentypen (2)

Zweistellige Bäume, Binärbäume

```
data Baum' = Blatt String
           | Wurzel' Baum' Int Bool Baum'
           deriving (Eq,Ord,Show)
```

Veranschaulichung: Werte vom Typ Baum':

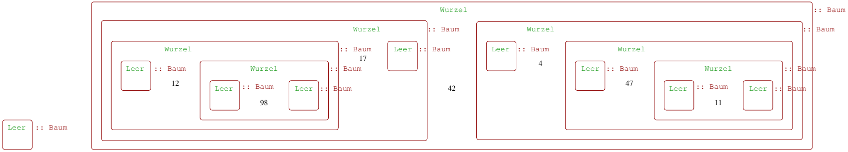


Beispiele rekursiver Summentypen (3)

Werte der Typen `Baum` und `Baum'`, zum Vergleich auf einer Seite:

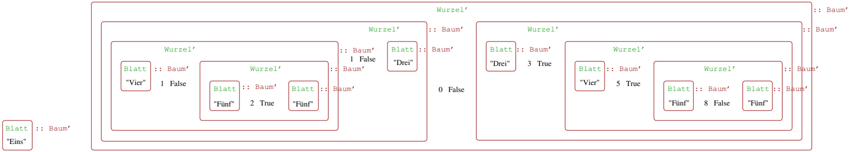
Werte vom Typ `Baum`

```
data Baum
  = Leer
  | Wurzel Baum Int Baum
  deriving (Eq,Ord,Show)
```



Werte vom Typ `Baum'`

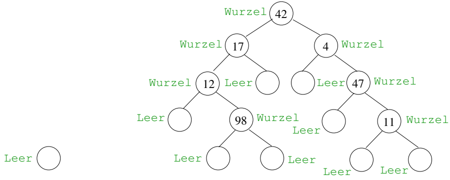
```
data Baum'
  = Blatt String
  | Wurzel' Baum' Int Bool Baum'
  deriving (Eq,Ord,Show)
```



Beispiele rekursiver Summentypen (4)

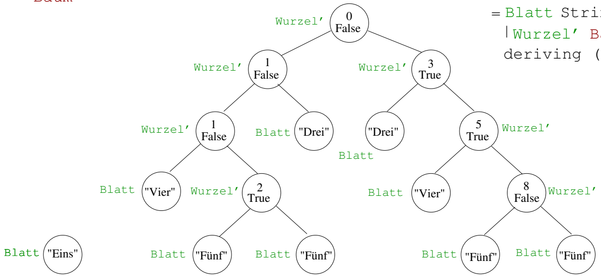
Werte d. Typen `Baum` und `Baum'`: Konventionelle Darstellung

Zwei Werte vom Typ
`Baum`




```
data Baum
  = Leer
  | Wurzel Baum Int Baum
  deriving (Eq,Ord,Show)
```

Zwei Werte vom Typ
`Baum'`




```
data Baum'
  = Blatt String
  | Wurzel' Baum' Int Bool Baum'
  deriving (Eq,Ord,Show)
```

Beispiele rekursiver Summentypen (5)

Dreistellige Bäume, Tertiärbäume

```
data TBaum = Nichts
           | Gabel Person TBaum TBaum TBaum
           deriving (Eq,Ord,Show)

data TBaum' = Laub Person [Anschrift]
            | Gabel' Person [Anschrift]
                     TBaum' TBaum' Tbaum'
            deriving (Eq,Ord,Show)
```

n-stellige Bäume

```
data NBaum      = NB Int [NBaum]
                  deriving (Eq,Ord,Show)

data NBaum'     = NB' (Person,[Anschrift]) [NBaum']
                  deriving (Eq,Ord,Show)

data Nadelbaum = Nb String Int Char Baum TBaum
                    [Nadelbaum] deriving...
```

Beispiele rekursiver Summentypen (6)

Suchbäume

```
type Schluessel  = Int
type Information = String
data Suchbaum    = Sb Schluessel Information
                   | Sk Schluessel Information
                          Suchbaum Suchbaum
                   deriving (Eq,Ord,Show)
```

Beispiele rekursiver Summentypen (7)

Kartei

```
type Soz_Vers_Nr = Int
type Schluessel  = Soz_Vers_Nr
type Str         = Strasse
type Hnr         = Hausnr
type Info        = (Person,
                     (Land -> Gemeinde -> [(Str,Hnr)]))

data Kartei = Kb Schluessel Info
            | Kk Schluessel Info Kartei Kartei
            deriving (Eq,Ord,Show)
                        -- Kb für Karteiblatt.
                        -- Kk für Karteikasten.
```

Beispiele wechselw. rekursiver Summentypen

Bürgernetzwerk von Freunden und verwandten und bekannten Nachbarn

```
type Verwandt = Bool
type Bekannt  = Bool
data Wohnform = EFH | ZFH | MFH | DH | RH | HH | PH
                deriving (Eq,Show)

data Buerger  = B Person Wohnform Nachbarn Freunde
                deriving (Eq,Show)
data Nachbarn = N [(Buerger,Verwandt,Bekannt)]
                deriving (Eq,Show)
data Freunde  = F [Buerger]
                deriving (Eq,Show)
```

Kapitel 5.1.4

Allgemeines Muster

Allgemeines Muster

...algebraischer Datentypdefinitionen:

```
data Typename = Con_1 t_11 ... t_1k_1
              | Con_2 t_21 ... t_2k_2
                ...
              | Con_n t_n1 ... t_nk_n
```

Sprechweisen:

- `Typename`: Freigewählter frischer Typname
- `Con_i`: Freigewählte frische (Datenwert-) Konstruktornamen
- `k_i`: Stelligkeit des Konstruktors `Con_i`
- `t_ij`: Namen existierender Typen

Beachte: Typname und Konstruktoren müssen stets mit einem Großbuchstaben beginnen (siehe z.B. `Bool`, `True`, `False`)!

(Datenwert-) Konstruktoren

...können als Funktionsdefinitionen gelesen werden:

```
Con_i t_i1 ... t_ik_i -> Typename
```

Die Konstruktion von Werten eines algebraischen Datentyps erfolgt durch Anwendung eines Konstruktors auf Werte "passenden" Typs:

```
v_i1 :: t_i1 ... v_ik_i :: t_ik_i
Con_i v_i1 ... v_ik_i :: Typename
```

Beispiele:

```
P "Adam" "Riese" Maennlich :: Person
A "Wien" "Karlsplatz" 13 "Austria" :: Anschrift
E_Buch "Simon Thompson" "Haskell" "Pearson" 2018
                          :: BildSchriftUndTontraeger
Haustier Katze "Garfield" 3.14 True :: Sammelsurium
```

Kapitel 5.1.5

Zusammenfassung

Art u. Anzahl der (Datenwert-) Konstruktoren

...algebraischer Datentypen liefern in Haskell den Schlüssel zu

- Aufzählungstypen
- Produkttypen
- Summentypen

Hinweis: Die Bezeichnungen Produkt- und Summentyp sind Grund in der Gesamtheit von algebraischen Datentypen zu sprechen.

Kategorisierung

Aufzählungstypen gekennzeichnet durch

- ausschließlich nullstellige Konstruktoren.

Produkttypen gekennzeichnet durch

- genau einen Konstruktor, der nicht nullstellig ist.

Summentypen gekennzeichnet durch

- mehrere Konstruktoren, von denen mindestens einer nicht nullstellig ist.

Aufzählungstypen

...gekennzeichnet durch ausschließlich 0-stellige Konstruktoren:

Beispiele:

```
data ()          = ()                  -- Ein 0-st. Kon.
data Bool        = False | True        -- Zwei 0-st. Kon.
data Ordering    = LT | EQ | GT        -- Drei 0-st. Kon.
data Spielfarbe  = Karo | Herz
                 | Pik | Kreuz         -- Vier 0-st. Kon.
data Werktage    = Montag | Dienstag
                 | Mittwoch | Donnerstag
                 | Freitag             -- Fünf 0-st. Kon.
```

Wertbeispiele:

- `()` einziger (def.) Wert des Typs `()`.
- `False`, `True` einzige (def.) Werte des Typs `Bool`.
- `LT`, `EQ`, `GT` einzige (def.) Werte des Typs `Ordering`.
- ...

Produkttypen

...gekennzeichnet durch genau einen nicht 0-stelligen Konstruktor:

Beispiele:

```
data Person      = P Vorname Nachname Geschlecht
data Anschrift   = A Gemeinde Strasse Hausnr Land
data Einwohner   = E Land Gemeinde [Person]
data Wohnsitze   = W Land (Person -> [Anschrift])
data Gemeldet    = G (Land -> Gemeinde -> Strasse
                        -> Hausnr -> [Person])
```

Wertbeispiele:

```
adam = P "Adam" "Riese" Maennlich  :: Person
ada  = P "Ada" "Lovelace" Weiblich :: Person
E "Austria" "Wien" [adam,ada] :: Einwohner
W "Australia" ws :: Wohnsitze
ws = \p -> case p of
            adam -> [A "Perth" "Main St" 42 "Australia"]
            ada  -> [A "Sydney" "High St" 1 "Australia",
                     A "Adelaide" "1st Ave" 10 "Australia"]
```

Summentypen

...gekennzeichnet durch mehrere Konstruktoren, davon mindestens einer nicht 0-stellig:

Beispiele:

```
data [a]          = []           -- Kein gültiges Haskell;
                    | a : [a]     -- nur zur Illustration!
data Maybe a      = Nothing
                    | Just a
data Either a b = Left a
                    | Right b
```

Wertbeispiele:

```
[] :: []; [1,2,3] :: [Int]; [True,False,True] :: [Bool]
Nothing   :: Maybe a;    Just 42  :: Maybe Int
Just 'a'  :: Maybe Char, Just ada :: Maybe Person
Left 42   :: Either Int Char, Right 'a' :: Either Int Char
Left adam :: Either Person (Person -> [Anschrift])
Right ws  :: Either Person (Person -> [Anschrift])
```

Summentypen (fgs.)

Beispiel:

```
data BildSchriftUndTontraeger =
        Buch Autor Titel Verlag Auflage Lieferbar
        | E_Buch Autor Titel Verlag LizenzBisJahr
        | DVD Titel Hauptdarsteller Regisseur Sprachen
        | CD Kuenstler Titel Spieldauer
```

Wertbeispiele:

```
Buch "Richard Bird" "Thinking Functionally"
     "Cambridge University Press" 1 True
  :: BildSchriftUndTontraeger
E_Buch "Simon Thompson" "Haskell" "Pearson" 2018
  :: BildSchriftUndTontraeger
DVD "Der Pate" ["Marlon Brando","Al Pacino"]
    "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"]
  :: BildSchriftUndTontraeger
CD "Angelika Nebel" "Klaviersonaten" (1,1,48)
  :: BildSchriftUndTontraeger
```

Summentypen (fgs.)

Beispiel:

```
data Baum = Leer
          | Wurzel Baum Int Baum
          deriving (Eq,Ord,Show)

data TBaum' = Laub Person [Anschrift]
            | Gabel' Person [Anschrift]
                     TBaum' TBaum' Tbaum'
```

Wertbeispiele:

```
Leer :: Baum
Wurzel Leer 42 Leer :: Baum
Wurzel (Wurzel Leer 17 Leer) 42 Leer :: Baum
adrs = [A "Sydney" "High St" 1 "Australia",
        A "Adelaide" "1st Ave" 10 "Australia"] :: [Anschrift]
t1 = Laub ada adrs :: TBaum'
t2 = Laub adam [A "Perth" "Main St" 42 "Australia"] :: TBaum'
t3 = Gabel' (P "Haskell" "Curry" Maennlich) [] t1 t2 t1 :: TBaum'
t4 = Gabel' ada adrs t2 t3 t4 :: TBaum'      -- nicht endlich!
```

Zusammenfassung

...mit `data` ein einheitliches Sprachkonstrukt in Haskell für

- Aufzählungstypen, Produkttypen, Summentypen als Ausprägungen algebraischer Datentypen.
- oft unterschiedliche Sprachkonstrukte in anderen Sprachen, z.B. drei in Pascal (siehe Anhang D).

Aufzählungs- und Produkttypen auffassbar als

- Randfall oder Spezialfall von Summentypen.

Algebraische Datentypdeklarationen können

- rekursiv (z.B. `Baum, TBaum'`) und wechselweise rekursiv (z.B. `Buerger, Nachbarn`) aufeinander Bezug nehmen.
- rekursive Typen ermöglichen es, Werte potentiell nicht beschränkter Größe (z.B. `t4 :: TBaum'`) zu konstruieren.

Kapitel 5.2

Funktionen auf algebraischen Datentypen

Funktionen auf algebraischen Datentypen

...werden üblicherweise mittels Musterpassung (engl. pattern matching) definiert.

In der Folge präsentieren wir einige Beispiele.

Beispiel einer Funktion

...auf BildSchriftUndTonTraeger-Daten, die Selektorfunktion titel:

```
titel :: BildSchriftUndTonTraeger -> Titel
titel (Buch   aut tit verl aufl lieferb) = tit
titel (E_Buch aut tit verl lizenz)       = tit
titel (DVD t _ _ _) = t
titel (CD  _ t _)   = t

titel (Buch "Richard Bird" "Thinking Functionally"
     "Cambridge University Press" 1 True)
 ->> "Thinking Functionally" :: Titel
```

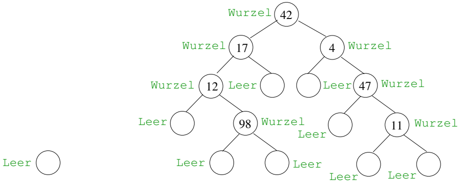
Aufrufbeispiele:

```
titel (E_Buch "Simon Thompson" "Haskell" "Pearson" 2018)
 ->> "Haskell" :: Titel
titel (DVD "Der Pate" ["Marlon Brando","Al Pacino"]
    "Francis Ford Coppola" ["Englisch","Deutsch","Italienisch"])
 ->> "Der Pate" :: Titel
titel (CD "Angelika Nebel" "Klaviersonaten" (1,1,48))
 ->> "Klaviersonaten" :: Titel
```

Binärbaumwerte: Konventionelle Darstellung

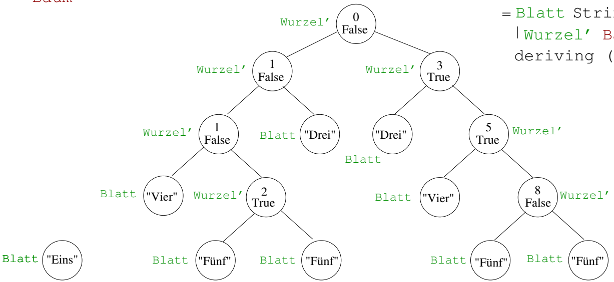
Erinnerung:

Zwei Werte vom Typ
Baum




```
data Baum
  = Leer
  | Wurzel Baum Int Baum
  deriving (Eq,Ord,Show)
```

Zwei Werte vom Typ
Baum'




```
data Baum'
  = Blatt String
  | Wurzel' Baum' Int Bool Baum'
  deriving (Eq,Ord,Show)
```

Beispiele für Funktionen auf Binärbäumen

...die Summenfkt. `summeMarken` und die Tiefenfkt. `tiefe`:

```
summeMarken :: Baum -> Int
summeMarken Leer = 0
summeMarken (Wurzel ltb n rtb)
     = n + summeMarken ltb + summMarken rtb

tiefe :: Baum' -> Int
tiefe (Blatt _) = 1
tiefe (Wurzel' ltb _ _ rtb)
    = 1 + max (tiefe ltb) (tiefe rtb)
```

Aufrufbeispiele:

```
summeMarken Leer ->>  0
summeMarken (Wurzel Leer 2 (Wurzel Leer 3 Leer)) ->> 5
tiefe (Blatt "Fun")  ->> 1
tiefe (Wurzel' (Blatt "Fun") 4 False
       (Wurzel' (Blatt "Prog") 11 True (Blatt ""))) ->> 3
```

Beispiele von Funktionen auf

...Bürgernetzwerkdaten, die Funktionen verwandteNachbarn und verwandteNachbarnNamen:

```
type Verwandt = Bool
type Bekannt  = Bool
data Wohnform = EFH | ZFH | MFH | DH | RH | HH | PH
                deriving (Eq,Ord,Show)

data Buerger  = B Person Wohnform Nachbarn Freunde
                deriving (Eq,Ord,Show)
data Nachbarn = N [(Buerger,Verwandt,Bekannt)]
                deriving (Eq,Ord,Show)
data Freunde  = F [Buerger]
                deriving (Eq,Ord,Show)

verwandteNachbarn :: Buerger -> [Person]
verwandteNachbarn (B _ _ (N ls) _)
 = [p | (B p _ _ _,verwandt,_) <- ls, verwandt == True]

verwandteNachbarnNamen :: Buerger -> [(Nachname,Vorname)]
verwandteNachbarnNamen (B _ _ (N ls) _)
 = [(nn,vn) | (B (P vn nn _) _ _ _,vw,_) <- ls, vw == True]
```

Kapitel 5.3

Feldsyntax

Transparente, sprechende Typdeklarationen

...drei Möglichkeiten bieten sich an, transparente und sprechende Datentypdeklarationen in Haskell zu erreichen:

- Kommentierung
- Typsynonyme
- Feldsyntax (Verbundtypsyntax)

  ...mit dem Zusatzvorteil

  - ‘geschenkter’ Selektorfunktionen
  - wesentlich vereinfachter weiterer Verarbeitungsfkt.

Transparente, sprechende Typdeklarationen

...mittels Kommentierung:

```
newtype Gb      = Gb (String,String,String)
                   deriving (Eq,Ord,Show)
data G          = M | W deriving (Eq,Ord,Show)

data Meldedaten = Md String  -- Vorname
                     String  -- Nachname
                     Gb      -- Geboren (tt,mm,jjjj)
                     G       -- Geschlecht (m/w)
                     String  -- Gemeinde
                     String  -- Strasse
                     Int     -- Hausnummer
                     Int     -- PLZ
                     String  -- Land
                   deriving (Eq,Ord,Show)
```

Transparente, sprechende Typdeklarationen

...mittels Typsynonymen:

```
type Vorname       = String
type Nachname      = String
type Ziffernfolge  = String
type Zf            = Ziffernfolge
newtype Gb         = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren       = Gb
data G             = M | W deriving (Eq,Ord,Show)
type Geschlecht    = G
type Gemeinde      = String
type Strasse       = String
type Hausnummer    = Int
type PLZ           = Int
type Land          = String

data Meldedaten    = Md Vorname Nachname Geboren
                        Geschlecht Gemeinde Strasse
                        Hausnummer PLZ Land deriving (Eq,Ord,Show)
```

Transparente, sprechende Typdeklarationen

...mittels Feldsyntax (Verbundtypsyntax):

```
type Ziffernfolge  = String
type Zf            = Ziffernfolge
data G             = M | W deriving (Eq,Ord,Show)
newtype Gb         = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten    = Md { vorname    :: String,
                          nachname   :: String,
                          geboren    :: Gb
                          geschlecht :: G,
                          gemeinde   :: String,
                          strasse    :: String,
                          hausnummer :: Int,
                          plz        :: Int,
                          land       :: String
                        } deriving (Eq,Ord,Show)
```

Transparente, sprechende Typdeklarationen

...typgleiche Felder können in der Feldsyntax durch Beistrich getrennt zusammengefasst werden:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data PersDaten = PD { vorname,
                      nachname,
                      gemeinde,
                      strasse,
                      land       :: String,
                      geboren    :: Gb,
                      geschlecht :: G,
                      hausnummer,
                      plz        :: Int
                    } deriving (Eq,Ord,Show)
```

Transparente, sprechende Typdeklarationen

...Feldnamen in Alternativen dürfen wiederholt werden, wenn ihr Typ für alle Vorkommen ident ist:

```
type Ziffernfolge = String
type Zf           = Ziffernfolge
data G            = M | W deriving (Eq,Ord,Show)
newtype Gb        = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten   = Md { vorname,
                         nachname,
                         gemeinde,
                         strasse,
                         land          :: String,
                         geboren       :: Gb
                         geschlecht    :: G,
                         hausnummer,
                         plz           :: Int
                       }
                    | KurzMd { vorname,
                               nachname :: String
                             } deriving (Eq,Ord,Show)
```

Transparente, sprechende Typdeklarationen

...mittels Kommentar, Typsynonymen und Feldsyntax:

```
type Vorname       = String
type Nachname      = String
type Ziffernfolge  = String
type Zf            = Ziffernfolge
newtype Gb         = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)
type Geboren       = Gb
data G             = M | W deriving (Eq,Ord,Show)
type Geschlecht    = G
type Gemeinde      = String
type Strasse       = String
type Hausnummer    = Int
type PLZ           = Int
type Land          = String
```

```
data Meldedaten = Md { vorname    :: Vorname,
                       nachname   :: Nachname,
                       geboren    :: Geboren, -- (tt,mm,jjjj)
                       geschlecht :: Geschlecht,
                       gemeinde   :: Gemeinde,
                       strasse    :: Strasse,
                       hausnummer :: Hausnummer,
                       plz        :: PLZ,
                       land       :: Land
                     } deriving (Eq,Ord,Show)
```

Selektorfunktionen

...über Kommentierung bzw. Typsynonyme definierte Typen erfordern üblicherweise musterbasiert-definierte Selektor-, Wertsetzungs- und Werterzeugungsfunktionen:

```
gibVorname :: Meldedaten -> Vorname
gibVorname (Md vn _ _ _ _ _ _ _ _)  = vn

gibNachname :: Meldedaten -> Nachname
gibNachname (Md _ nn _ _ _ _ _ _ _) = nn

...

gibPLZ :: Meldedaten -> PLZ
gibPLZ (Md _ _ _ _ _ _ _ plz _) = hsnr

gibLand :: Meldedaten -> Land
gibLand (Md _ _ _ _ _ _ _ _ land)   = land
```

...für Meldedaten sind auf diese Weise 9 Selektorfunktionen separat zu schreiben.

Wertsetzungsfunktionen

...in gleicher Weise gilt dies für Wertsetzungsfunktionen:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
setzeVorname vn (Md _ nn geb gs gem str hsnr plz land)
 = Md vn nn geb gs gem str hsnr plz land

setzeNachname :: Nachname -> Meldedaten -> Meldedaten
setzeNachname nn (Md vn _ geb gs gem str hsnr plz land)
 = Md vn nn geb gs gem str hsnr plz land

...

setzePLZ :: PLZ -> Meldedaten -> Meldedaten
setzePLZ plz (Md vn nn geb gs gem str hsnr _ land)
 = Md vn nn geb gs gem str hsnr plz land

setzeLand :: Land -> Meldedaten -> Meldedaten
setzeLand land (Md vn nn geb gs gem str hsnr plz _)
 = Md vn nn geb gs gem str hsnr plz land
```

...9 Wertsetzungsfunktionen für Meldedaten.

Werterzeugungsfunktionen

...ebenso für Werterzeugungsfunktionen:

```
undefiniert = undefiniert -- Auswertung terminiert nicht!

erzeugeMdMitVorname :: Vorname -> Meldedaten
erzeugeMdMitVorname vorname
 = Md vorname undefiniert undefiniert undefiniert
      undefiniert undefiniert undefiniert undefiniert
      undefiniert

...

erzeugeMdMitLand :: Land -> Meldedaten
erzeugeMdMitVorname land
 = Md undefiniert undefiniert undefiniert undefiniert
      undefiniert undefiniert undefiniert undefiniert
      land)
```

...9 Werterzeugungsfunktionen für Meldedaten.

Selektorfunktionen bei Feldnamenverwendung

...die Selektorfunktionen sind durch die Feldnamen gegeben und damit 'geschenkt':

```
gibVorname :: Meldedaten -> Vorname
gibVorname = vorname

gibNachname :: Meldedaten -> Nachname
gibNachname = nachname

...

gibPLZ :: Meldedaten -> PLZ
gibPLZ = plz

gibLand :: Meldedaten -> Land
gibLand = land
```

Anmerkung: Die Funktionen `gibVorname`, `gibNachname`, etc., sind *de facto* nur Synonyme bzw. Aliase der Feldnamen `vorname`, `nachname`, etc.; ihre Einführung deshalb im Grunde obsolet.

Wertsetzungsfkt. bei Feldnamenverwendung

...Feldnamen erlauben eine wesentlich knappere Schreibweise der Wertsetzungsfunktionen:

```
setzeVorname :: Vorname -> Meldedaten -> Meldedaten
setzeVorname vn md = md {vorname = vn}

setzeNachname :: Nachname -> Meldedaten -> Meldedaten
setzeNachname nn md = md {nachname = nn}

...

setzePLZ :: PLZ -> Meldedaten -> Meldedaten
setzePLZ p md = md {plz = p}

setzeLand :: Land -> Meldedaten -> Meldedaten
setzeLand ld md = md {land = ld}
```

Werterzeugungsfkt. bei Feldnamenverwendung

...dies gilt vergleichbar auch für Werterzeugungsfunktionen:

```
erzeugeMdMitVorname :: Vorname -> Meldedaten
erzeugeMdMitVorname vn = Md vorname = vn

erzeugeMdMitNachname :: Nachname -> Meldedaten
erzeugeMdMitNachname nn = Md nachname = nn

...

erzeugeMdMitPLZ :: PLZ -> Meldedaten
erzeugeMdMitPLZ p = Md plz = p

erzeugeMdMitLand :: Land -> Meldedaten
erzeugeMdMitLand ld = Md land = ld
```

...nicht genannte Felder werden automatisch 'undefiniert' gesetzt.

Setzen oder initialisieren mehrerer Felder

...auch mehrere Felder können gesetzt werden:

```
setzeVorundNachname :: Vorname -> Nachname
                       -> Meldedaten -> Meldedaten
setzeVorundNachname vn nn md
 = md {vorname=vn, nachname=nn}
            -- Nicht genannte Felder behalten ihren Wert.
...

erzeugeMdMitVorundNachname :: Vorname -> Nachname
                              -> Meldedaten
erzeugeMdMitVorundNachname vn nn
 = Md {vorname=vn, nachname=nn}
   -- Nicht genannte Felder werden ‘undefiniert’ gesetzt.
```

Weitere Beispiele von Feldnamenverwendungen

...liefere Vor- und Nachnamen, getrennt durch ein Leerzeichen:

```
gibVollerName :: Meldedaten -> String
gibVollerName md
 = vorname md ++ " " ++ nachname md

gibVollerName' :: Meldedaten -> String
gibVollerName' (Md {vorname = vn, nachname = nn})
 = vn ++ " " ++ nn
```

Gleichwertig, doch weniger bequem ohne Feldnamen:

```
gibVollerName'' :: Meldedaten -> String
gibVollerName'' (Md vn nn _ _ _ _ _ _ _)
 = vn ++ " " ++ nn
```

Ausblick

...in Kapitel 10 werden wir über

- monomorphe (Daten-) Typen und Funktionen

hinausgehend

- polymorphe (Daten-) Typen

und

- polymorphe und überladene Operationen und Funktionen

besprechen.

Kapitel 5.4

Anwendungshinweise

Kapitel 5.4.1

Produkttypen vs. Tupeltypen

Produkttypen vs. Tupeltypen

...am Beispiel des Typs Person als Produkt- und als Tupeltyp:

Produkttyp

```
data Person = P Vorname Nachname Geschlecht
  -- Produkttyp im engeren Sinn: Konstruktor P mehrstellig
data Person = P (Vorname,Nachname,Geschlecht)
  -- Produkttyp im weiteren Sinn: Konstruktor P einstellig
newtype Person = P (Vorname,Nachname,Geschlecht)
   -- Kein Produkttyp im strengen Sinn: newtype statt data
```

Tupeltyp

```
type Person = (Vorname, Nachname, Geschlecht)
```

Offensichtlicher Unterschied: Kein Konstruktor im Tupeltyp von Person wie im entsprechenden Produktyp, hier P.

Vorteile von Produkt- gegenüber Tupeltypen

...zusammengefasst in einem Wort: Typsicherheit.

- Werte des Produkttyps sind typgesichert, da sie mit dem mit dem Konstruktor "markiert" sind.
- 'Zufällig' passende Werte sind deshalb nicht irrtümlich, versehentlich oder absichtlich als Wert des Produkttyps manipulierbar: Typsicherheit! (Vgl. frühere Beispiele zu Wertpapier-, Wasserstands- und Ortsdaten).
- Aussagekräftigere (Typ-) Fehlermeldungen; Typsynonyme können wg. Expansion in Fehlermeldungen fehlen.

Vorteile von Tupel- gegenüber Produkttypen

...zusammengefasst in einem Wort: Anwendungskomfort.

- Auf den Grundtypen (von Typsynonymen) und Tupeln vordefinierte Funktionen stehen ohne Einschränkung zur Verfügung (z.B. `fst, snd, (+), (*),...`).
- Tupelwerte erfordern keine Konstruktoren und sind deshalb (geringfügig) kompakter (weniger Schreibaufwand für den Programmquelltext).
- (Geringfügig) höhere Ausführungsperformanz, da "ein-" und "auspacken" von Tupelwerten entfällt.

  Hinweis: Bei einstelligen Produkttypen ist statt einer `data`- auch eine `newtype`-Deklaration möglich; hier kein Effizienzverlust, da der Konstruktor nur zur Übersetzungszeit für die Überprüfung der Typkorrektheit benötigt wird.

Beispiel: Telefonbuch

...mittels type-Deklarationen ausschließlich:

```
type Vorname     = String
type Nachname    = String
type Spitzname   = String
type Name        = (Vorname,Nachname)
type Telefonnr   = Int
type Telefonbuch = [(Name,Telefonnr)]

gibTelnr :: Name -> Telefonbuch -> Telefonnr
gibTelnr name ((name',tnr):tb_rest)
 | name == name' = tnr
 | otherwise     = gibTelnr name tb_rest
gibTelnr _ []    = error "Telefonnummer unbekannt"

gibSpitzname :: Telefonnr -> Telefonbuch -> Spitzname
gibSpitzname tnr (((vn,_),tnr'):tb_rest)
 | tnr == tnr'    = vn++"ilein"
 | otherwise      = gibSpitzname tnr tb_rest
gibSpitzname _ [] = error "Spitzname unbekannt"
```

Beispiel: Telefonbuch

...mittels newtype-Deklarationen ausschließlich:

```
newtype Vorname     = Vn String deriving (Eq,Show)
newtype Nachname    = Nn String deriving (Eq,Show)
newtype Spitzname   = Sn String deriving (Eq,Show)
newtype Name        = N (Vorname,Nachname) deriving (Eq,Show)
newtype Telefonnr   = T Int deriving (Eq,Show)
newtype Telefonbuch = Tb [(Name,Telefonnr)] deriving (Eq,Show)

gibTelnr :: Name -> Telefonbuch -> Telefonnr
gibTelnr (N name) (Tb ((N name',T tnr):tb_rest))
 | name == name'   = T tnr
 | otherwise       = gibTelnr (N name) (Tb tb_rest)
gibTelnr _ (Tb []) = error "Telefonnummer unbekannt"

gibSpitzname :: Telefonnr -> Telefonbuch -> Spitzname
gibSpitzname (T tnr) (Tb ((N (Vn vn,_),T tnr'):tb_rest))
 | tnr == tnr'         = Sn (vn++"ilein")
 | otherwise           = gibSpitzname (T tnr) (Tb tb_rest)
gibSpitzname _ (Tb []) = error "Spitzname unbekannt"
```

Vergleich der `type`- und `newtype`-Varianten

`type`-Deklarationen:

- Die ‘eigentlichen’ Werte (der Typen `String`, `Int`) liegen frei zutage und können direkt in Mustern bezeichnet werden.
- Ergebnisse können unmittelbar zurückgegeben werden.

`newtype`-Deklarationen:

- Typkonstruktoren (`Vn`, `Nn`, `Sn`, `N`, `T`, `Tb`) sind integraler Bestandteil von Werten und müssen deshalb explizit in Argumentmustern angegeben werden, um die ‘eigentlichen’ Werte (der Typen `String`, `Int`) freizulegen und bezeichnen zu können.
- Bei der Rückgabe von Ergebnissen muss der ‘eigentliche’ Wert (der Typen `String`, `Int`) in den passenden Konstruktor ‘eingepackt’ werden (in `gibTelnr`: ‘`T tnr`’ statt ‘`tnr`’; in `gibSpitzname`: ‘`Sn vn++"ilein"`’ statt ‘`vn++"ilein"`’).

Kapitel 5.4.2

Typsynonyme vs. neue Typen

Eigenschaften von `newtype`-Deklarationen

`newtype`-Deklarationen entsprechen im Hinblick auf

- Typsicherheit `data`-Deklarationen: Datenwerte liegen geschützt und markiert hinter `newtype`-Konstruktoren.
- Performanz `type`-Deklarationen: `newtype`-Konstruktoren werden nur zur Übersetzungszeit für die Typüberprüfung benötigt; nicht zur Laufzeit.

`newtype`-Deklarationen vereinen somit die

- besten Eigenschaften von `data`- und `type`-Deklarationen.

Aber: Typsicherheit ohne Zusatzaufwand zur Laufzeit hat einen Preis!

Beschränkungen von newtype-Deklarationen

newtype-Deklarationen sind beschränkt auf Deklarationen mit

- genau einem (Datenwert-) Konstruktor mit genau einem (Daten-) Feld ("there is no free lunch"!).

Beispiele:

```
                                   1 Feld
newtype Person = P (Vorname, Nachname, Geboren)
     1 1-stelliger Konstruktor              -- Möglich!

                                3 Felder
newtype Person = P Vorname Nachname Geboren
     1 3-stelliger Konstruktor  -- Nicht möglich!
```

Kapitel 5.4.3

Faustregel zur Wahl von `type`, `newtype`, `data`

Faustregel zur Wahl von `type`-Deklarationen

`type`-Deklarationen führen einen

- neuen Namen für einen existierenden Typ ein.

Sind sinnvoll, wenn

- durch 'sprechendere' Typnamen die Transparenz und Verständlichkeit von Signaturen erhöht werden soll.
- auf den Komfort, die auf dem Grundtyp definierten Funktionen weiterzubenutzen, nicht verzichtet werden soll.

Allerdings:

- Keine höhere Typsicherheit.

Faustregel z. Wahl v. `newtype`-Deklarationen

`newtype`-Deklarationen führen einen

- neuen Typ für einen existierenden Typ ein.

Sind sinnvoll, wenn

- zusätzlich zu 'sprechenderen' Typnamen auch höhere Typsicherheit erreicht werden soll.
- 'Erhöhte' Laufzeitkosten algebraischer Typen vermieden werden sollen (und können).
- Typen zu Instanzen von Typklassen gemacht werden sollen (siehe Kapitel 4.3 und Kapitel 11.4).

Allerdings:

- Eingeschränkte Anwendungsmöglichkeit. Nur möglich für Typen mit genau einem Datenwertkonstruktor und genau einem Datenfeld.

Faustregel zur Wahl von `data`-Deklarationen

`data`-Deklarationen erlauben in freier Weise

- neue Typen zu kreieren.

Sind sinnvoll (bzw. nötig), wenn

- Typsicherheit benötigt wird.
- eine newtype-Deklaration ausscheidet, weil ein neuer bislang nicht existierender Typ mit mehr als einem Konstruktor oder mehr als einem Datenfeld benötigt wird.

Allerdings:

- Leicht erhöhte Verarbeitungskosten gegenüber `newtype`-Deklarationen durch Konstruktorbehandlung nicht zur Übersetzungs-, sondern auch zur Laufzeit.

Summa summarum

`type`- vs. `newtype`- und `data`-Deklarationen:

...`type`-Deklarationen

- wo 'angemessen' und 'ausreichend', zusätzliche Typsicherheit nicht erforderlich ist.
- `newtype`- und `data`-Deklarationen, wo zusätzliche Typsicherheit nötig und unverzichtbar ist.

`newtype`- vs. `data`-Deklarationen:

...`newtype`-Deklarationen

- wo möglich; `data`-Deklarationen, wo nötig.

Kapitel 5.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 5 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 8, Benutzerdefinierte Datentypen)
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 4, Algebraische Datentypen)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 8.1, Type declarations; Kapitel 8.2, Data declarations; Kapitel 8.3, Newtype declarations; Kapitel 8.4, Recursive types)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 5 (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 7, Making our own Types and Type Classes; Kapitel 12, Monoids – Wrapping an Existing Type into a New Type)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 12, Konstruktion von Datenstrukturen)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 6, Typen; Kapitel 8, Polymorphe und abhängige Typen; Kapitel 9, Spezifikationen und Typklassen)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 5 (3)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions; Kapitel 3, Defining Types, Streamlining Functions – Defining a New Data Type, Type Synonyms, Algebraic Data Types)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 14, Algebraic types)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14, Algebraic types)

Kapitel 6

Muster und mehr

Muster und mehr

Muster, Musterpassung (Kap. 6.1)

- Elementare Datentypen
- Tupeltypen
- Listentypen
  - `[]`-Muster
  - `(p:ps)`-Muster, `(p:(q:qs))`-Muster, etc.
  - als-Muster (engl. as pattern)
- Algebraische Datentypen

Listenkomprehension (Kap. 6.2)

Konstruktoren, Operatoren (Kap. 6.3)

- Begriffsbestimmung und Vergleich am Beispiel von Listen

Kapitel 6.1

Muster, Musterpassung

Muster, Musterpassung

Muster sind

- (syntaktische) Ausdrücke zur Beschreibung der Struktur von Werten.

Musterpassung (engl. pattern matching) erlaubt

- in Funktionsdefinitionen mithilfe einer Folge von Mustern Alternativen auszuwählen. Dabei werden die Muster in einer festen Reihenfolge (von oben nach unten) durchprobiert; passt die Struktur eines (Argument-) Werts auf ein Muster, wird diese Alternative ausgewählt.

Kapitel 6.1.1

Muster für Werte elementarer Datentypen

Muster für Werte elementarer Datentypen (1)

...am Beispiel von Funktionen auf Wahrheitswerten:

Konstanten und Joker als Muster:

```
nicht :: Bool -> Bool
nicht True  = False
nicht _     = True

und :: Bool -> Bool -> Bool
und True True = True
und _ _       = False

oder :: Bool -> Bool -> Bool
oder False False = False
oder _ _         = True
```

Muster für Werte elementarer Datentypen (2)

Konstanten, Variablen und Joker als Muster:

```
nund :: Bool -> Bool -> Bool
nund True True = False
nund _ _       = True

noder :: Bool -> Bool -> Bool
noder False False = True
noder _ _         = False

xoder :: Bool -> Bool -> Bool
xoder a b = a /= b

wennDannSonst :: Bool -> a -> a -> a
wennDannSonst True  t _ = t
wennDannSonst False _ e = e
```

Muster für Werte elementarer Datentypen (3)

...am Beispiel von Funktionen auf ganzen Zahlen:

Konstanten, Variablen und Joker als Muster:

```
mult :: Int -> Int -> Int
mult _ 0 = 0
mult 0 _ = 0
mult m 1 = m
mult 1 n = n
mult m n = m * n

potenz :: Integer -> Integer -> Integer
potenz _ 0 = 1
potenz m n = m * potenz m (n-1)
```

Zusammenfassung

Muster für Werte elementarer Datentypen sind:

- Konstanten: `0`, `42`, `3.14`, `'c'`, `True`,...
  ⇝ ein Wert passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `b`, `m`, `n`, `t`, `e`,...
  ⇝ jeder Wert passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  ⇝ jeder Wert passt (aber ist rechtsseitig nicht verwendbar).

Kapitel 6.1.2

Muster für Werte von Tupeltypen

Muster für Werte von Tupeltypen

...am Beispiel von polymorphen Funktionen und Funktionen auf ganzen Zahlen:

Konstanten, Variablen und Joker als Muster:

```
fst' :: (a,b,c) -> a
fst' (x,_,_) = x

snd' :: (a,b,c) -> b
snd' (_,y,_) = y

thd' :: (a,b,c) -> c
thd' (_,_,z) = z

binom' :: (Integer,Integer) -> Integer
binom' (n,k)
 | k==0 || n==k = 1
 | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

Zusammenfassung

Muster für Werte von Tupeltypen sind:

- Konstanten: `(0,0)`, `(0,"Null")`, `(3.14,"pi",True)`,...
  ⇝ ein Wert passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `t`, `t1`,...
  ⇝ jeder Wert passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  ⇝ jeder Wert passt (aber ist rechtsseitig nicht verwendbar).
- Kombinationen aus Konstanten, Variablen, Jokern:
  `(m,n)`, `(True,n,_)`, `(_,(m,_,n),3.14,k,_)`,...
  ⇝ ein Wert passt, wenn er strukturell mit dem Muster übereinstimmt.

Kapitel 6.1.3

Muster für Werte von Listentypen

Muster für Werte von Listentypen (1)

Konstanten als Muster; Konstruktormuster mit Konstanten, Variablen und Jokern:

```
sum :: [Int] -> Int
sum []     = 0
sum (0:xs) = sum xs
sum (x:xs) = x + sum xs

mult :: [Int] -> Int
mult []     = 1
mult (0:_)  = 0
mult (1:xs) = mult xs
mult (x:xs) = x * mult xs
```

Muster für Werte von Listentypen (2)

Konstanten, Variablen und Joker als Muster; Konstruktormuster mit Jokern:

```
kopf :: [a] -> a
kopf (x:_)  =  x

rest :: [a] -> [a]
rest (_:xs) = xs

leer :: [a] -> Bool
leer []     = True
leer _      = False

verbinde :: [a] -> [a] -> [a] -> [a]
verbinde ps qs rs = ps ++ qs ++ rs
```

Muster für Werte von Listentypen (3)

Konstanten und Joker als Muster; Konstruktormuster mit Variablen und Jokern:

```
nimm :: Int -> [a] -> [a]   -- entspricht vordef.
nimm m ys = case (m,ys) of          -- Fkt. take
             (0,_)      -> []
             (_,[])     -> []
             (n,(x:xs)) -> x : nimm (n - 1) xs

streiche :: Int -> [a] -> [a] -- entspricht vordef.
streiche m ys = case (m,ys) of      -- Fkt. drop
             (0,_)      -> ys
             (_,[])     -> []
             (n,(_:xs)) -> streiche (n - 1) xs
```

Muster für Werte von Listentypen (4)

Konstruktormuster erlauben auch, “endlich tief” in eine Liste hineinzusehen:

```
maxElem :: Ord a => [a] -> a
maxElem []         = error "Ungueltige Eingabe"
maxElem (y:[])     = y
maxElem (x:y:ys) = maxElem ((max x y) : ys)
```

Muster für Werte von Listentypen (5)

Konstanten, Variablen und Joker als Muster für Zeichenreihenwerte; Konstruktormuster mit Variablen für Zeichenreihen:

```
anfuegen :: String -> String -> String
anfuegen "" t = t
anfuegen s "" = s
anfuegen s t  = s ++ t

istPrefix :: String -> String -> Bool
istPrefix "" _           = True
istPrefix (c:_) ""       = False
istPrefix (c:cs) (d:ds) = (c==d) && istPrefix cs ds
```

Zusammenfassung (1)

Muster für Werte von Listentypen, speziell Zeichenreihen, sind:

- Konstanten: `[]`, `""`, `[1,2,3]`, `[1..50]`, `['a'..'z']`, `[True,False,True,True]`, `"aeiou"`,...
  $\leadsto$ ein Wert passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `p`, `q`, `ps`, `qs`...
  $\leadsto$ jeder Wert passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  $\leadsto$ jeder Wert passt (aber ist rechtsseitig nicht verwendbar).

Zusammenfassung (2)

- Konstruktormuster:
  (⟨muster_listenkopf⟩:⟨muster_listenrest⟩),
  (p:ps), (p:q:qs),...
  ⇝ ein Listenwert *ls* passt auf das Konstruktormuster (⟨muster_listenkopf⟩:⟨muster_listenrest⟩), wenn ⟨muster_listenkopf⟩ und ⟨muster_listenrest⟩ gültige Musterausdrücke für Listenköpfe und Listenreste sind, *ls* nicht leer ist, der Kopf von *ls* strukturell mit ⟨muster_listenkopf⟩ übereinstimmt und der Rest von *ls* mit ⟨muster_listenrest⟩.

  ls passt strukturell auf das Konstruktormuster (p:ps) bzw. (p:q:qs) mit p, ps, q, qs Variablenmuster, wenn *ls* nicht leer ist bzw. mindestens 2 Elemente enthält.

Kapitel 6.1.4

Muster für Werte algebraischer Datentypen

Muster für Werte algebraischer Datentypen (1)

Konstanten entsprechend 0-stelligen Konstruktoren als Muster:

```
type Zeichenreihe = [Char]
data Jahreszeiten = Fruehling | Sommer
                  | Herbst | Winter

wetter :: Jahreszeiten -> Zeichenreihe
wetter Fruehling = "Launisch"
wetter Sommer    = "Sonnig"
wetter Herbst    = "Windig"
wetter Winter    = "Frostig"
```

Muster für Werte algebraischer Datentypen (2)

Konstruktormuster mit Variablen:

```
type Zett = Int
data Ausdruck = Opd Zett
               | Add Ausdruck Ausdruck
               | Sub Ausdruck Ausdruck
               | Quad Ausdruck

eval :: Ausdruck -> Zett
eval (Opd n)      = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
eval (Quad e)     = (eval e)^2
```

Muster für Werte algebraischer Datentypen (3)

Konstanten als Muster; Konstruktormuster mit Variablen und Jokern:

```
type Zett = Int
data Baum a b = Blatt a
              | Wurzel b (Baum a b) (Baum a b)

tiefe :: (Baum a b) -> Zett
tiefe (Blatt _)       = 1
tiefe (Wurzel _ l r) = 1 + max (tiefe l) (tiefe r)


data Liste a = Leer | Kopf a (Liste a)

listenLaenge :: Liste a -> Zett
listenLaenge Leer         = 0
listenLaenge (Kopf _ xs) = 1 + listenLaenge xs
```

Muster für Werte algebraischer Datentypen (4)

Konstruktormuster erlauben wie für Listen auch in Werte algebraischer Datentypen "endlich tief" hineinzusehen:

```
type Zett = Int
data Baum = Blatt Zett
          | Gabel Zett Baum Baum

putzig :: Baum -> Zett
putzig (Blatt 7) = 42
putzig (Blatt n) = n*n
putzig (Gabel n (Blatt m) (Gabel p (Blatt q) (Blatt r)))
                 = n+m+p+q+r
putzig (Gabel n (Gabel _ (Blatt q) (Blatt r)) (Blatt _))
                 = n*(q+r)
putzig _         = 0
```

Zusammenfassung

Muster für Werte algebraische Typen sind:

- Konstanten: `Sommer`, `Winter`, `Empty`,...
  ⇝ ein Wert passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `e`, `e1`, `e2`, `t`,...
  ⇝jeder Wert passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  ⇝ jeder Wert passt (aber ist rechtsseitig nicht verwendbar).
- Konstruktormuster: `(Opd e)`, `(Add e1 e2)`, `(Blatt' 7)`, `(Blatt' n)`, `(Blatt' _)`, `(Gabel 42 l r)`, (Kopf _ hs),...
  ⇝ ein Wert passt strukturell auf das Konstruktormuster, wenn seine Struktur mit der des Musters übereinstimmt.

Kapitel 6.1.5

Das als-Muster

Das als-Muster (1)

Oft sehr nützlich ist das sog. als-Muster (engl. as pattern).

Betrachte folgendes Beispiel:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe (c:cs)
  = (c:cs) : nichtleerePostfixe cs
nichtleerePostfixe _ = []

nichtleerePostfixe "Curry"
  ->> ["Curry","urry","rry","ry","y"]
```

Die rechte Seite der ersten definierenden Gleichung nimmt Bezug auf

- das gesamte strukturierte Argument: `(c:cs)`
- einen Teil des strukturierten Arguments: `cs`

Das als-Muster (2)

Das als-Muster erlaubt dies einfacher auszudrücken:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe s@(_:cs)
                    = s : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```

Das als-Muster `s@(_:cs)` (`@` gelesen als "als" (engl. "as")) bietet je einen Namen an für

- das gesamte Argument, in diesem Beispiel: `s`
- für die relevanten strukturellen Komponenten des Arguments, in diesem Beispiel für den Rest der Liste, wenn die Argumentliste nicht leer ist: `cs`

Vorteile aus der Verwendung des als-Musters

...anhand des Beispiels der Funktion `nichtleerePostfixe`:

- Mittels `s` lässt sich auf das gesamte Argument Bezug nehmen; mittels `cs` auf die strukturelle Komponente des Listenrests, wenn die Argumentliste nicht leer ist.
- Die Verwendung des als-Musters führt deshalb wie in diesem Beispiel meist zu einfacheren und übersichtlicheren Definitionen.

Zum Vergleich

...beide Definitionen noch einmal gegenübergestellt:

- Mit als-Muster:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe s@(_:cs)
                          = s : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```

- Ohne als-Muster:

```
nichtleerePostfixe :: String -> [String]
nichtleerePostfixe (c:cs)
  = (c:cs) : nichtleerePostfixe cs
nichtleerePostfixe _ = []
```

Weitere Beispiele (1)

...Listen und als-Muster.

Die Funktion `listTransform` mit `als`-Muster:

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : l) ++ xs
```

Zum Vergleich `listTransform` ohne `als`-Muster:

```
listTransform :: [a] -> [a]
listTransform (x:xs) = (x : (x : xs)) ++ xs
```

Weitere Beispiele (2)

...Tupel und als-Muster.

Die Funktion tausche mit als-Muster:

```
tausche :: Eq a => (a,a) -> (a,a)
tausche p@(c,d)
 | c /= d    = (d,c)
 | otherwise = p
```

Zum Vergleich tausche ohne als-Muster:

```
tausche :: Eq a => (a,a) -> (a,a)
tausche (c,d)
 | c /= d    = (d,c)
 | otherwise = (c,d)
```

Weitere Beispiele (3)

...Tupel und als-Muster.

Die Funktion `tauscheBedingt` mit `als`-Muster:

```
tauscheBedingt :: (a,Bool,a) -> (a,Bool,a)
tauscheBedingt t@(b,c,d)
 | c     = (d,c,b)
 | not c = t
```

Zum Vergleich `tauscheBedingt` ohne `als`-Muster:

```
tauscheBedingt :: (a,Bool,a) -> (a,Bool,a)
tauscheBedingt (b,c,d)
 | c     = (d,c,b)
 | not c = (b,c,d)
```

Generell

...ist das als-Muster über Listen und Tupel hinaus allgemein für algebraische Datentypen mit strukturierten Werten nützlich.

Kapitel 6.1.6

Zusammenfassung

Vorteile musterbasierter Funktionsdefinitionen

Musterbasierte Funktionsdefinitionen

- sind elegant.
- führen (i.a.) zu knappen, gut lesbaren Spezifikationen.

Zur Illustration: Die Funktion `binom'` mit Mustern sowie ohne Muster mittels Standardselektoren:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)     -- mit Mustern
 | k==0 || n==k = 1
 | otherwise    = binom' (n-1,k-1) + binom' (n-1,k)
```

```
binom' :: (Integer,Integer) -> Integer
binom' p         -- ohne Muster mit Std.-Selektoren
  | snd(p)==0 || snd(p)==fst(p) = 1
  | otherwise = binom' (fst(p)-1,snd(p)-1)
                + binom' (fst(p)-1,snd(p))
```

Allerdings

...musterbasierte Funktionsdefinitionen können

- zu subtilen Fehlern führen.
- Programmänderungen/-weiterentwicklungen erschweren, "bis hin zur Tortur", etwa beim Hinzukommen eines oder mehrerer weiterer Parameter.

  (siehe dazu: Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003, S. 164.)

Kapitel 6.2

Listenkomprehension

Listenkomprehension (1)

...ein charakteristisches, elegantes und ausdruckskräftiges Sprachmittel

- funktionaler Programmiersprachen

ohne Parallele in anderen Paradigmen, das die Mengenbildungsoperation aus der Mathematik auf Listen nachbildet.

Listenkomprehension (2)

...erlaubt Listen auf eine Weise zu beschreiben, in der ihre Elemente durch

- filtern, testen und transformieren der Elemente anderer Listen

erzeugt werden.

Zur Illustration: Eine Reihe von Beispielen zur Verwendung von Listenkomprehension in

- Ausdrücken
- Funktionsdefinitionen
- Zeichenreihen (als speziellen Listen)

Listenkomprehension in Ausdrücken (1)

Zwei Listen:

```
lst1 = [1,2,3,4]
lst2 = [1,2,4,7,8,11,12,42]
```

Ein Generator, eine Transformation:

```
[ 3*n | n <- lst1 ]
 ->> [3,6,9,12]

[ square n | n <- lst2 ]
 ->> [1,4,16,49,64,121,144,1764]

[ isPrime n | n <- lst2 ]
 ->> [False,True,False,True,False,True,False,False]
```

Listenkomprehension in Ausdrücken (2)

Ein Generator, ein bzw. zwei Tests, eine Transformation:

```
[ fac n | n <- lst2, isPowOfTwo n ]
 ->> [1,2,24,40320]

[ id n | n <- lst2, isPowOfTwo n, n>=5 ]     -- ","
 ->> [8]                               -- steht für "und"
```

Zwei Generatoren, ein Filter, zwei Tests, eine Transformation:

```
[ ((m,n),m+n) | m <- lst1, n <- tail lst2,
                m<=2, n<=7 ]
  ->> [((1,2),3),((1,4),5),((1,7),8),
       ((2,2),4),((2,4),6),((2,7),9)]
```

Listenkomprehension in Ausdrücken (3)

Zwei Generatoren, zwei Filter, ein Test, eine Transformation:

```
[fib ((+) m n) | m <- take 3 lst1, n <- drop 5 lst2,
                 (odd (m+n) || (m*n)>20) ]
 ->> [fib ((+) m n) | m <- [1,2,3], n <- [11,12,42],
                      (odd (m+n) || (m*n)>20) ]
 ->> [fib (1+42),fib (2+11),fib (3+12),fib (3+42)]
 ->> [fib 43, fib 13, fib 15, fib 45]
 ->> ...
```

Listenkomprehension in Fkt.-Definitionen (1)

Abstandsberechnung vom Ursprung einer Liste von Punkten:

```
type Point = (Float,Float)

distanceFromOrigin :: [Point] -> [Float]
distanceFromOrigin plst
      = [ sqrt (squ x + squ y) | (x,y) <- plst ]

distanceFromOrigin [(3.0,4.0),(1.0,1.0),(-1.0,3.0)]
 ->> [5.0,1.414,3.1623]
```

Test auf Ungradheit und Gradheit aller Listenelemente:

```
allOdd :: [Integer] -> Bool
allOdd xs = ([ x | x <- xs, isOdd x ] == xs)

allOdd [2..22] ->> False

allEven :: [Integer] -> Bool
allEven xs = ([ x | x <- xs, isOdd x ] == [])

allEven [2,4..22] ->> True
```

Listenkomprehension in Fkt.-Definitionen (2)

```
grabCapVowels :: String -> String
grabCapVowels s = [ c | c <- s, isCapVowel c ]

isCapVowel :: Char -> Bool
isCapVowel 'A' = True
isCapVowel 'E' = True
isCapVowel 'I' = True
isCapVowel 'O' = True
isCapVowel 'U' = True
isCapVowel  _  = False

grabCapVowels "Alles Eint Informatik Ohne Unterlass!"
               ->> "AEIOU"
```

Listenkomprehension in Fkt.-Definitionen (3)

QuickSort:

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (x:xs) = quickSort [ y | y <- xs, y <= x ]
                   ++ [x]
                   ++ quickSort [ y | y <- xs, y > x ]
```

Anmerkung: Funktionsanwendung bindet stärker als Listenkonstruktion; deshalb Klammerung des Musters `(x:xs)` in der zweiten definierenden Gleichung `quickSort (x:xs) = ...`

Listenkomprehension und Zeichenreihen

...Zeichenreihen sind in Haskell ein Typalias für Listen von Zeichen:

```
type String = [Char]
```

Beispiel:

```
"Haskell" == ['H','a','s','k','e','l','l']
```

Für Zeichenreihen als spezielle Listen stehen deshalb dieselben

- Funktionen
- Komprehensionsmechanismen

zur Verfügung wie für allgemeine Listen.

Listenfunktionen für Zeichenreihen

Beispiele für Funktionen auf Zeichenreihen als Listen:

```
"Haskell"!!3 ->> 'k'

take 5 "Haskell" ->> "Haske"

drop 5 "Haskell" ->> "ll"

length "Haskell" ->> 7

zip "Haskell" [1,2,3] ->> [('H',1),('a',2),('s',3)]
```

Listenkomprehension für Zeichenreihen

Zählen der Kleinbuchen in einer Zeichenreihe:

```
lowers :: String -> Int
lowers xs = length [ x | x <- xs, isLower x ]

lowers "Haskell" ->> 6
```

Zählen der Vorkommen eines bestimmten Zeichens in einer Zeichenreihe:

```
count :: Char -> String -> Int
count c xs = length [ x | x <- xs, x == c ]

count ’s’ "Mississippi" ->> 4
```

Kapitel 6.3

Konstruktoren, Operatoren

Konstruktoren vs. Operatoren

Konstruktoren führen zu eindeutigen Darstellungen von Werten, Operatoren nicht.

Beispiel:

- (:) ist (einziger) Konstruktor auf Listen.
- (++) ist (einer von vielen) Operator(en) auf Listen.

Betrachte:

```
[42,17,4] == (42:(17:(4:[])))   -- Eindeutige Dar-
                             -- stellung von [42,17,4]
                         -- mittels des Konstruktors (:).

[42,17,4] == [42,17] ++ [] ++ [4]   -- Viele Darstel-
          == [42] ++ [17,4] ++ []          -- lungen von
          == [42] ++ [] ++ [17,4]          -- [42,17,4]
          == ...                          -- mittels des
                                   -- Operators (++).
```

Bemerkungen

- Die Darstellung `(42:(17:(4:[])))` deutet an, dass eine Liste ein Objekt ist; erzwungen durch die Typstruktur.
- Anders in imperativen/objektorientierten Sprachen: Listen sind dort nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

Operatoren in Mustern nicht zulässig

Aufgrund der fehlenden Zerlegungseindeutigkeit bei Verwendung von Listenoperatoren dürfen

- Listenoperatoren nicht in Mustern

verwendet werden!

Veranschaulichung

```
deleteTwo :: (Char,Char) -> String -> String
deleteTwo _ ""      = ""
deleteTwo _ (s:[]) = [s]
deleteTwo (c,d) (s:(t:ts))
 | (c,d) == (s,t)  = deleteTwo (c,d) ts
 | otherwise       = s : deleteTwo (c,d) (t:ts)
```

...ist sinnvoll und zulässig, weil das Muster `(s:(t:ts))` die Struktur "passender" Argumentwerte und darmit das Resultat eindeutig festlegt.

```
deleteTwo :: (Char,Char) -> String -> String
deleteTwo _ ""      = ""
deleteTwo _ (s:[]) = [s]
deleteTwo (c,d) s@([s1]++[s2])
 | [c,d] == [s1]   = deleteTwo (c,d) s2
 | otherwise       = head s : deleteTwo (c,d) tail s
```

...ist nicht sinnvoll und unzulässig, weil das "Muster `([s1]++[s2])`" die Wahl von `[s1]` und `[s2]` zur Zerlegung von `s` nicht eindeutig festlegt und sich je nach Zerlegung ein anderes Resultat ergäbe.

Kapitel 6.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 6 (1)

- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 4.2, List operations)
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 5.1.4, Automatische Erzeugung von Listen)
- Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.4, List comprehensions)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 4.4, Pattern matching; Kapitel 5, List comprehensions)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 6 (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 3, Syntax in Functions – Pattern Matching)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 12, Barcode Recognition – List Comprehensions)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 13, Mehr syntaktischer Zucker)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.4, Lists; Kapitel 4.1, Lists)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 6 (3)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 5.4, Lists in Haskell; Kapitel 5.5, List comprehensions; Kapitel 7.1, Pattern matching revisited; Kapitel 7.2, Lists and list patterns; Kapitel 9.1, Patterns of computation over lists; Kapitel 17.3, List comprehensions revisited)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 5.5, Lists in Haskell; Kapitel 5.6, List comprehensions; Kapitel 7.1, Pattern matching revisited; Kapitel 7.2, Lists and list patterns; Kapitel 10.1, Patterns of computation over lists; Kapitel 17.3, List comprehensions revisited)

Teil III

Applikative Programmierung

Applikatives Programmieren

...im strengen Sinn:

- Applikatives Programmieren ist ein Programmieren auf dem Niveau von elementaren Werten.
- Mit Konstanten, Variablen und Funktionsapplikationen werden Ausdrücke gebildet, deren Werte stets elementar sind.
- Durch explizite Abstraktion nach gewissen Variablen erhält man Funktionen.

Damit:

- Tragendes Konzept applikativer Programmierung zur Programmerstellung ist die Funktionsapplikation, d.h. die Anwendung von Funktionen auf (elementare) Werte.

Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 1.

Funktionales Programmieren

...im strengen Sinn:

- Funktionales Programmieren ist ein Programmieren auf Funktionsniveau.
- Ausgehend von Funktionen werden mit Hilfe von Funktionen höherer Ordnung neue Funktionen gebildet.
- Es treten im Programm keine Applikationen von Funktionen auf elementare Werte auf.

Damit:

- Tragendes Konzept funktionaler Programmierung zur Programmerstellung ist die Bildung neuer Funktionen aus gegebenen Funktionen mit Hilfe von Funktionen höherer Ordnung.

Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 1.

Kapitel 7

Rekursion

Kapitel 7.1

Motivation

Rekursion

Zentrales Mittel funktionaler Sprachen

- Wiederholungen auszudrücken (Beachte: Wir haben keine Anweisungen und deshalb auch keine Schleifen in funktionalen Sprachen).

Rekursives Vorgehen

- führt oft auf sehr elegante Lösungen, die konzeptuell wesentlich einfacher und intuitiver sind als schleifenbasierte imperative Lösungen (Typische Beispiele: Quicksort, Türme von Hanoi).

Rekursion insgesamt so wichtig, dass

- eine Klassifizierung von Rekursionstypen zweckmäßig ist.

...eine solche Klassifizierung nehmen wir in der Folge vor.

Quicksort, Türme von Hanoi

...Beispiele, für die rekursives Vorgehen auf besonders

- intuitive, einfache und elegante Lösungen

führt.

- Quicksort: Zum schnellen Sortieren.
- Türme von Hanoi: Bezeichnet die auf eine hinterindische Sage zurückgehende Aufgabe einer Gruppe von Mönchen, die seit dem Anbeginn der Zeit damit beschäftigt sind, einen Turm aus 50 goldenen Scheiben nach festen Regeln umzuschichten.*

* Die Sage berichtet, dass das Ende der Welt gekommen ist, wenn die Mönche ihre Aufgabe vollendet haben.

Quicksort

...bereits besprochen:

```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (n:ns) = quickSort smaller
                   ++ [n]
                   ++ quickSort larger
                   where
                    smaller = [m | m<-ns, m<=n]
                    larger  = [m | m<-ns, m>n]
```

Türme von Hanoi

Ausgangssituation:

Gegeben sind drei Stapel(plätze) A, B und C. Auf Platz A liegt ein Stapel paarweise verschieden großer Scheiben, die mit von unten nach oben abnehmender Größe aufgeschichtet sind.

Aufgabe:

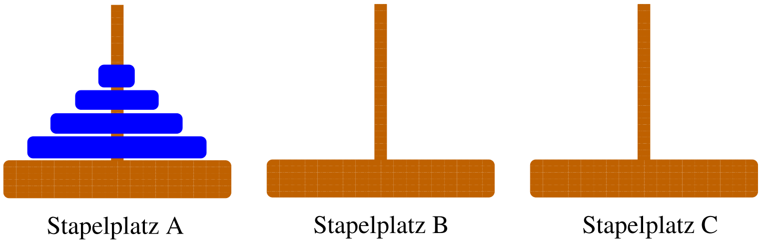
Schichte den Scheibenstapel von Platz A auf Platz C um unter Zuhilfenahme von Platz B.

Randbedingung:

Es darf stets nur eine Scheibe bewegt werden; es darf nie eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze zu liegen kommen.

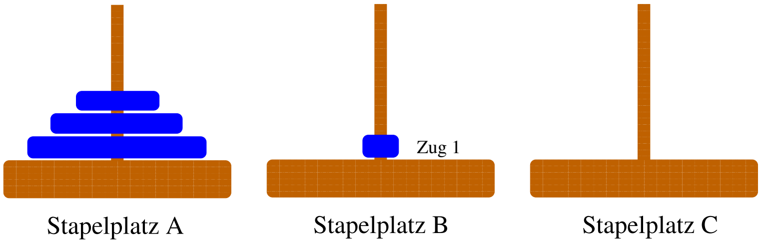
Türme von Hanoi (1)

Ausgangssituation:



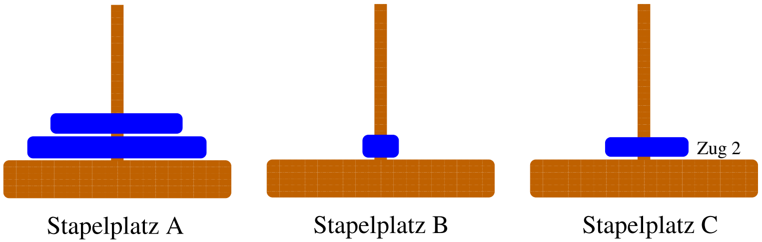
Türme von Hanoi (2)

Nach einem Zug:



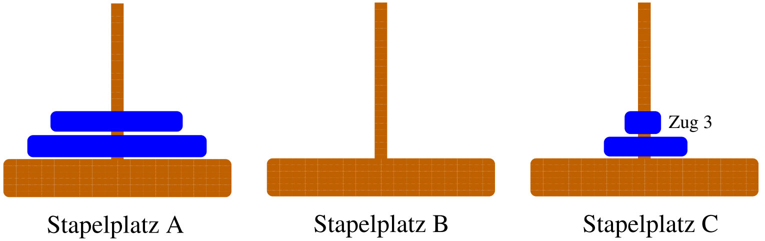
Türme von Hanoi (3)

Nach zwei Zügen:



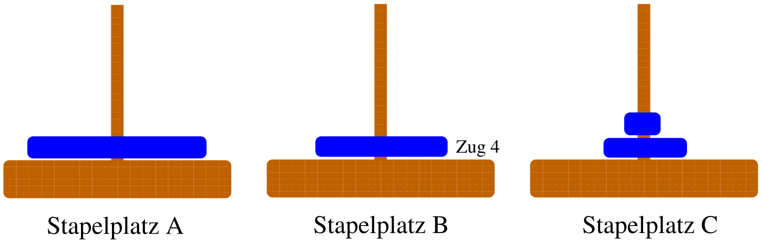
Türme von Hanoi (4)

Nach drei Zügen:



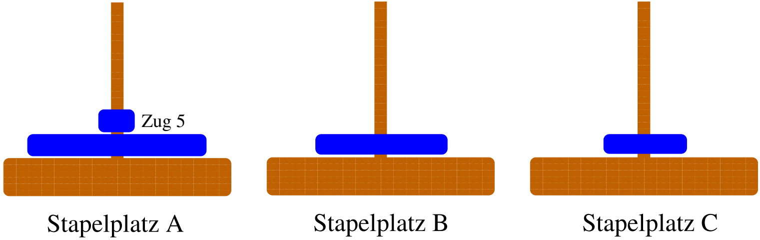
Türme von Hanoi (5)

Nach vier Zügen:



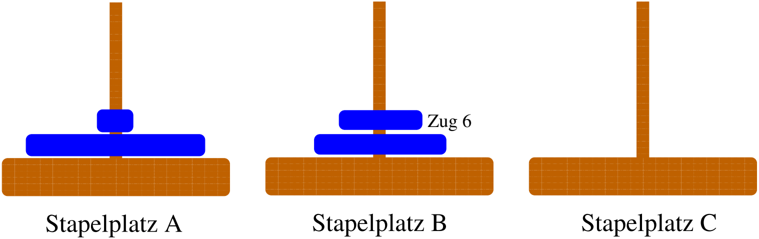
Türme von Hanoi (6)

Nach fünf Zügen:



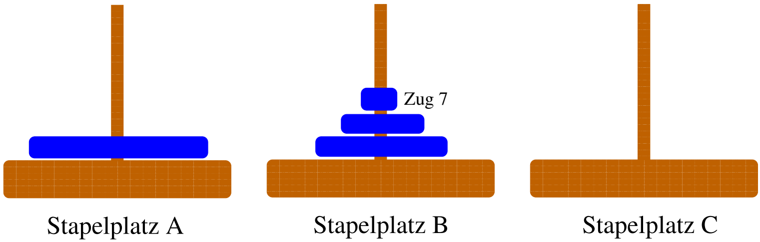
Türme von Hanoi (7)

Nach sechs Zügen:



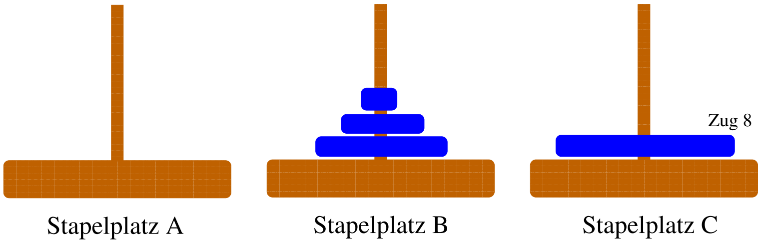
Türme von Hanoi (8)

Nach sieben Zügen:



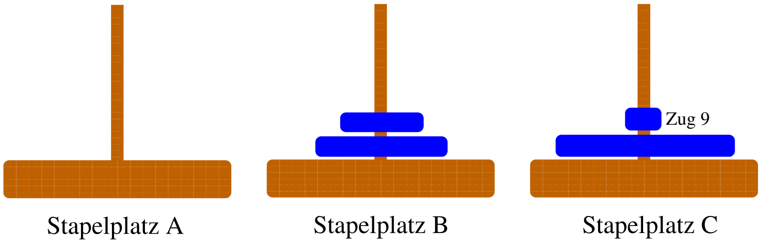
Türme von Hanoi (9)

Nach acht Zügen:



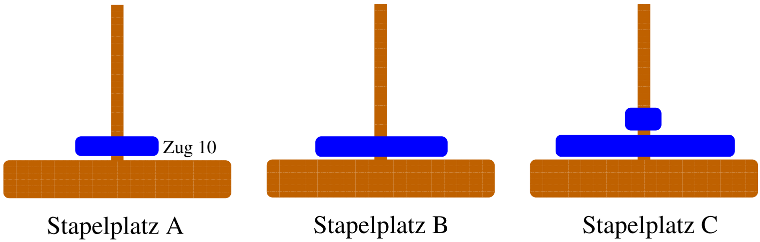
Türme von Hanoi (10)

Nach neun Zügen:



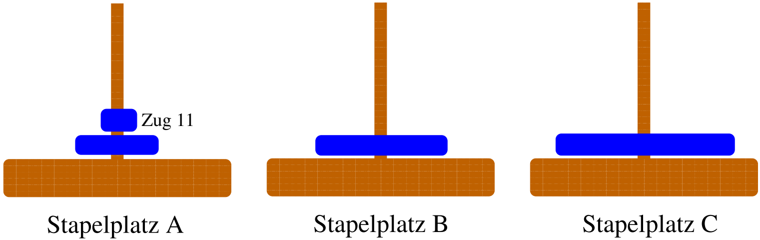
Türme von Hanoi (11)

Nach zehn Zügen:



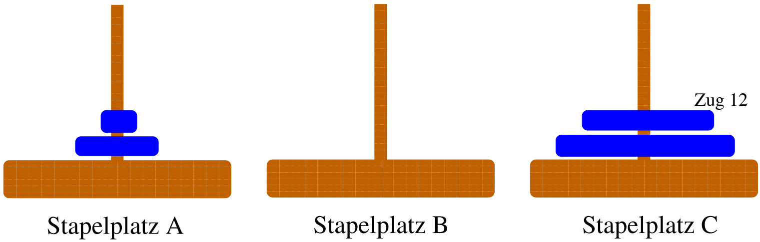
Türme von Hanoi (12)

Nach elf Zügen:



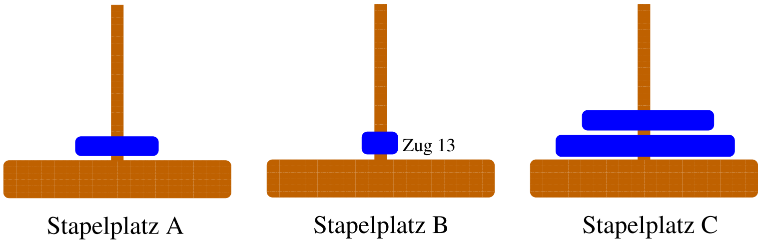
Türme von Hanoi (13)

Nach zwölf Zügen:



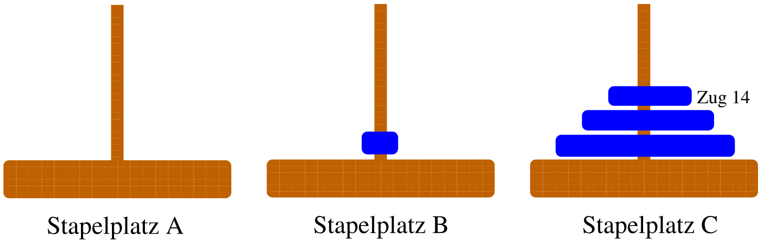
Türme von Hanoi (14)

Nach dreizehn Zügen:



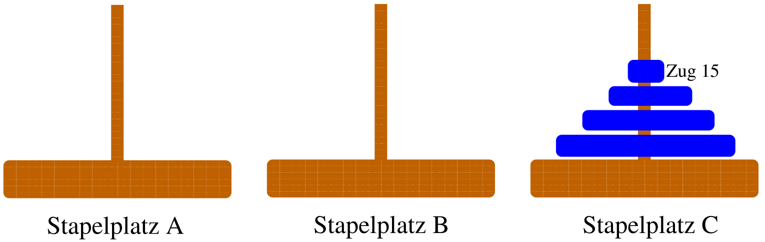
Türme von Hanoi (15)

Nach vierzehn Zügen:



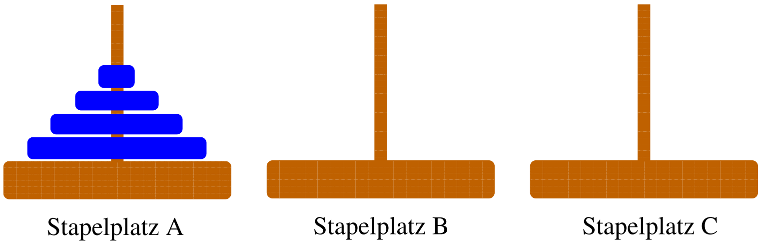
Türme von Hanoi (16)

Nach fünfzehn Zügen:



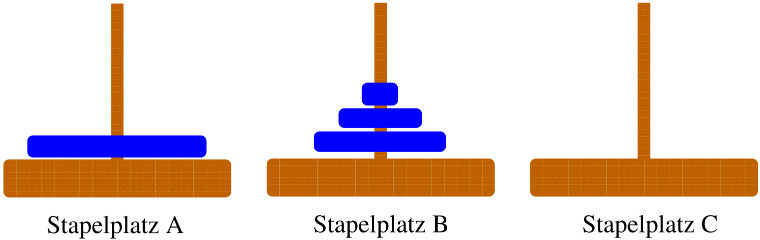
Veranschaulichung der Rekursionsidee (1)

Aufgabe: Verschiebe Turm $[1, 2, \ldots, N]$ von Ausgangsstapel A nach Zielstapel C unter Verwendung von Hilfsstapel B.



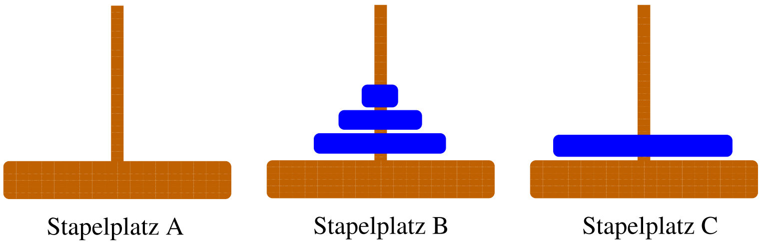
Veranschaulichung der Rekursionsidee (2)

Schritt 1: Platz schaffen & freispielen: Verschiebe Turm $[1, 2, \ldots, N-1]$ von Ausgangsstapel A nach Hilfsstapel B:



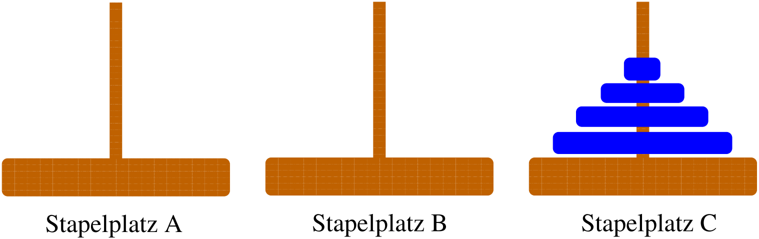
Veranschaulichung der Rekursionidee (3)

Schritt 2: Freigespielt, jetzt wird gezogen: Verschiebe Scheibe N von Ausgangsstapel A nach Zielstapel C:



Veranschaulichung der Rekursionsidee (4)

3) Aufräumen: Verschiebe Turm $[1, 2, \ldots, N-1]$ von Hilfsstapel B nach Zielstapel C:



Türme von Hanoi: Die Rekursionsidee

Um einen Turm $[1, 2, \ldots, N-1, N]$ aus N Scheiben, dessen kleinste Scheibe mit 1, dessen größte mit N bezeichnet sei, von Stapel A nach Stapel C unter Zuhilfenahme von Stapel B zu verschieben,

1) verschiebe den Turm $[1, 2, \ldots, N-1]$ aus $N-1$ Scheiben von A nach B unter Zuhilfenahme von Stapel C
2) verschiebe die nun frei liegende unterste Scheibe N von A nach C
3) verschiebe den Turm $[1, 2, \ldots, N-1]$ aus $N-1$ Scheiben von B nach C unter Zuhilfenahme von Stapel A

Türme von Hanoi: Implementierung in Haskell

```
type Turmhoehe   = Int   -- Anzahl Scheiben
type Scheiben_Nr = Int   -- Scheibenidentifikator
type A_Stapel    = Char  -- Ausgangsstapel A
type Z_Stapel    = Char  -- Zielstapel Z
type H_Stapel    = Char  -- Hilfsstapel H

hanoi :: Turmhoehe -> A_Stapel -> Z_Stapel -> H_Stapel
                   -> [(Scheiben_Nr,A_Stapel,Z_Stapel)]
hanoi n a z h
 | n==0      = []            -- Nichts zu tun, fertig
 | otherwise =               -- Sonst: 3 Schritte
    (hanoi (n-1) a h z)      -- (N-1)-Turm von A nach H über Z
    ++ [(n,a,z)]             -- Scheibe N von A nach Z
    ++ (hanoi (n-1) h z a)   -- (N-1)-Turm von H nach Z über A
```

Türme von Hanoi: Aufrufe der Funktion hanoi

```
Main>hanoi 1 'A' 'C' 'B'
[(1,'A','C')]

Main>hanoi 2 'A' 'C' 'B'
[(1,'A','B'),(2,'A','C'),(1,'B','C')]

Main>hanoi 3 'A' 'C' 'B'
[(1,'A','C'),(2,'A','B'),(1,'C','B'),(3,'A','C'),
(1,'B','A'),(2,'B','C'),(1,'A','C')]

Main>hanoi 4 'A' 'C' 'B'
[(1,'A','B'),(2,'A','C'),(1,'B','C'),(3,'A','B'),
(1,'C','A'),(2,'C','B'),(1,'A','B'),(4,'A','C'),
(1,'B','C'),(2,'B','A'),(1,'C','A'),(3,'B','C'),
(1,'A','B'),(2,'A','C'),(1,'B','C')]
```

Kapitel 7.2

Rekursionstypen

Klassifikation der Rekursionstypen

Eine Rechenvorschrift heißt rekursiv, wenn

- sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Wir unterscheiden Rekursion auf

- mikroskopischer Ebene
  ...betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe.
- makroskopischer Ebene
  ...betrachtet Systeme von Rechenvorschriften und ihre wechselseitigen Aufrufe.

Rekursion auf mikroskopischer Ebene (1)

...folgende Unterscheidungen und Sprechweisen sind üblich:

1. Repetitive (schlichte, endständige) Rekursion
   $\rightsquigarrow$ pro Zweig höchstens ein rekursiver Aufruf und zwar stets als äußerste Operation.

   Beispiel:

```
ggt :: Integer -> Integer -> Integer
ggt m n
 | n == 0  = m                    -- Zweig 1
 | m >= n  = ggt (m-n) n          -- Zweig 2
 | m < n   = ggt (n-m) m          -- Zweig 3
```

Rekursion auf mikroskopischer Ebene (2)

2. Lineare Rekursion
   ⇝ pro Zweig höchstens ein rekursiver Aufruf, davon mindestens einmal nicht als äußerste Operation.

   Beispiel:

```
powerThree :: Integer -> Integer
powerThree n
 | n == 0   = 1                      -- Zweig 1
 | n > 0    = 3 * powerThree (n-1)  -- Zweig 2
```

   Beachte: Im Zweig 2, $n > 0$ ist "$*$" die äußerste Operation, nicht `powerThree`!

Rekursion auf mikroskopischer Ebene (3)

3. Baumartige (kaskadenartige) Rekursion
   $\rightsquigarrow$ pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen.

   Beispiel:

```
binom' :: (Integer,Integer) -> Integer
binom' (n,k)
 | k==0 || n==k = 1                          -- Zweig 1
 | otherwise    =
    binom' (n-1,k-1) + binom' (n-1,k) -- Zweig 2
```

Rekursion auf mikroskopischer Ebene (4)

4. Geschachtelte Rekursion
   $\rightsquigarrow$ rekursive Aufrufe enthalten rekursive Aufrufe als Argumente.

   Beispiel:

   ```
   fun91 :: Integer -> Integer
   fun91 n
    | n >  100 = n - 10
    | n <= 100 = fun91 (fun91 (n+11))
   ```

   Übungsaufgabe: Warum heißt die Funktion wohl `fun91`?

Rekursion auf mikroskopischer Ebene (5)

...auf mikroskopischer Ebene unterscheiden wir:

- Repetitive (schlichte, endständige) Rekursion
- Lineare Rekursion
- Baumartige (kaskadenartige) Rekursion
- Geschachtelte Rekursion

...zusammengefasst unter dem gemeinsamen Oberbegriff:

- Rekursion (genauer: direkte Rekursion)

Rekursion auf makroskopischer Ebene

Indirekte (verschränkte, wechselweise) Rekursion

$\rightsquigarrow$ zwei oder mehr Funktionen rufen sich wechselweise auf.

Beispiel:

```
isOdd :: Integer -> Bool
isOdd n
 | n == 0 = False
 | n > 0  = isEven (n-1)

isEven :: Integer -> Bool
isEven n
 | n == 0 = True
 | n > 0  = isOdd (n-1)
```

Eleganz, Effizienz, Implementierung

Viele Probleme sind rekursiv besonders

- elegant zu lösen (z.B. Quicksort, Türme von Hanoi)
- jedoch nicht immer unmittelbar effizient ($\neq$ effektiv!) (z.B. die naive Berechnung der Fibonacci-Zahlen)
  - Gefahr: (Unnötige) Mehrfachberechnungen
  - Besonders anfällig: Baum-/kaskadenartige Rekursion

Aus Implementierungssicht ist

- repetitive Rekursion am (kosten-) günstigsten.
- geschachtelte Rekursion am ungünstigsten.

Die Folge der Fibonacci-Zahlen

...die unendliche Folge der Fibonacci-Zahlen $f_0, f_1, f_2, \ldots$ ist in folgender Weise definiert:

$$f_0 = 0, \;\; f_1 = 1 \quad \text{und} \quad f_n = f_{n-1} + f_{n-2} \quad \text{für alle } n \geq 2$$

Anfang der Folge der Fibonacci-Zahlen:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Naive Berechnung der Fibonacci-Zahlen (1)

Die naheliegende, unmittelbar an die Definition angelehnte naive Implementierung mit baumartiger Rekursion zur Berechnung der Fibonacci-Zahlen:

```
fib :: Integer -> Integer
fib n
 | n == 0     = 0
 | n == 1     = 1
 | otherwise  = fib (n-1) + fib (n-2)
```

...ist sehr, seeehr langsaaaaaaam (ausprobieren!)

Naive Berechnung der Fibonacci-Zahlen (2)

Veranschaulichung der Ineffizienz durch manuelle Auswertung:

```
fib 0 ->> 0                        --  1 Aufrufe von fib

fib 1 ->> 1                        --  1 Aufrufe von fib

fib 2 ->> fib 1 + fib 0
      ->> 1 + 0
      ->> 1                        --  3 Aufrufe von fib

fib 3 ->> fib 2 + fib 1
      ->> (fib 1 + fib 0) + 1
      ->> (1 + 0) + 1
      ->> 2                        --  5 Aufrufe von fib
```

Naive Berechnung der Fibonacci-Zahlen (3)

```
fib 4 ->> fib 3 + fib 2
      ->> (fib 2 + fib 1) + (fib 1 + fib 0)
      ->> ((fib 1 + fib 0) + 1) + (1 + 0)
      ->> ((1 + 0) + 1) + (1 + 0)
      ->> 3                        --  9 Aufrufe von fib

fib 5 ->> fib 4 + fib 3
      ->> (fib 3 + fib 2) + (fib 2 + fib 1)
      ->> ((fib 2 + fib 1) + (fib 1 + fib 0))
                     + ((fib 1 + fib 0) + 1)
      ->> (((fib 1 + fib 0) + 1)
                     + (1 + 0))+((1 + 0) + 1)
      ->> (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
      ->> 5                        -- 15 Aufrufe von fib
```

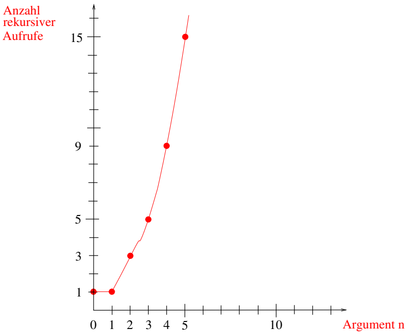
Naive Berechnung der Fibonacci-Zahlen (4)

```
fib 8 ->> fib 7 + fib 6
      ->> (fib 6 + fib 5) + (fib 5 + fib 4)
      ->> ((fib 5 + fib 4) + (fib 4 + fib 3))
          + ((fib 4 + fib 3) + (fib 3 + fib 2))
      ->> (((fib 4 + fib 3) + (fib 3 + fib 2))
           + (fib 3 + fib 2) + (fib 2 + fib 1)))
          + (((fib 3 + fib 2) + (fib 2 + fib 1))
           + ((fib 2 + fib 1) + (fib 1 + fib 0)))
      ->> ...
      ->> 21                        -- 60 Aufrufe von fib
```

Naive Berechnung der Fibonacci-Zahlen (5)

...die baumartig-rekursive naive Berechnung der Fibonacci-Zahlen führt zu äußerst vielen Mehrfachberechnungen.

Insgesamt wächst der Berechnungsaufwand exponentiell!



Effiziente Berechnung der Fibonacci-Z. (1)

Fibonacci-Zahlen lassen sich auf viele Arten effizient berechnen, z.B. durch

- Rechnen auf Parameterposition!

```
fib :: Integer -> Integer
fib n = fib' n 0 1
 where
  fib' :: Integer -> Integer -> Integer -> Integer
  fib' 0 a b = a
  fib' n a b = fib' (n-1) b (a+b)
```

Übungsaufgabe: Werten Sie die Funktion manuell für einige Werte aus, um zu sehen, wie die obige Implementierung von `fib` mithilfe von `fib'` die Fibonacci-Zahlen berechnet.

Effiziente Berechnung der Fibonacci-Z. (2)

...die i.w. gleiche Idee verteilt auf verschiedene Funktionen leistet folgendes System von Rechenvorschriften:

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n
 | n == 0    = (0,1)
 | otherwise = fibSchritt (fibPaar (n-1))

fib :: Integer -> Integer
fib n = fst (fibPaar n)
```

Übungsaufgabe: Werten Sie auch diese Funktion manuell für einige Werte aus, um zu sehen, wie die Berechnung der Fibonacci-Zahlen erfolgt und vergleichen Sie dies mit der vorigen Implementierung.

Effiziente Berechnung der Fibonacci-Z. (3)

...sog. Memo-Funktionen führen ebenfalls zu einer effizienten Implementierung, eine Idee, die auf Donald Michie zurückgeht:

- Donald Michie. 'Memo' Functions and Machine Learning. Nature 218:19-22, 1968.

```
memo_fib :: [Int]
memo_fib = [fib n | n <- [0..]]      -- Memo-Liste

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = memo_fib !! (n-1) + memo_fib !! (n-2)
```

Hinweis: Die Listenelementzugriffsfunktion `(!!)` hat den Typ `(!!) :: [a] -> Int -> a`; deshalb ist `fib` hier über `Int` definiert, nicht über `Integer`.

Abhilfe bei ungünstigem Rekursionsverhalten

...(oft) ist folgende Verbesserung möglich:

- Ersetzung ungünstiger durch günstigere Rekursionsmuster!

  Z.B. Rückführung linearer Rekursion auf repetitive Rekursion.

Rückführung linearer auf repetitive Rek. (1)

...am Beispiel der Fakultätsfunktion:

Naheliegende Implementierung mittels linearer Rekursion:

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1
 | otherwise = n * fac (n-1)
```

Rückführung linearer auf repetitive Rek. (2)

Günstigere Formulierung mittels repetitiver Rekursion durch

- Rechnen auf Parameterposition.

```
fac :: Integer -> Integer
fac n = fac_repetitiv n 1

fac_repetitiv :: Integer -> Integer -> Integer
fac_repetitiv n resultat
 | n == 0    = resultat
 | otherwise = fac_repetitiv (n-1) (n*resultat)
```

Beachte: Überlagerungen mit anderen Effekten sind möglich, so dass sich möglicherweise kein Effizienzgewinn realisiert!

Weitere Möglichkeiten zur Verbesserung

...bieten spezielle Programmiertechniken wie

- Dynamische Programmierung
- Memoization

Zentrale Idee:

- Speicherung und Wiederverwendung bereits berechneter (Teil-) Ergebnisse statt deren Wiederberechnung.

  (Siehe etwa die effiziente Berechnung der Fibonacci-Zahlen mithilfe einer Memo-Funktion)

Hinweis: Dynamische Programmierung und Memoization werden in der LVA 185.A05 Fortgeschrittene funktionale Programmierung ausführlich behandelt.

Kapitel 7.3

Aufrufgraphen

Struktur von Programmen

Programme funktionaler Programmiersprachen (auch Haskell-Programme) sind i.a.

- Systeme (wechselweiser) rekursiver Rechenvorschriften, die sich hierarchisch oder/und wechselweise aufeinander abstützen.

Aufrufgraphen erleichtern es, sich über die

- Struktur von Systemen von Rechenvorschriften

Klarheit zu verschaffen.

Aufrufgraphen

...sei S ein System von Rechenvorschriften.

Der Aufrufgraph von S enthält

- einen Knoten für jede in S deklarierte Rechenvorschrift
- eine gerichtete Kante vom Knoten f zum Knoten g genau dann, wenn im Rumpf der zu f gehörigen Rechenvorschrift die zu g gehörige Rechenvorschrift aufgerufen wird.

Beispiele: Die Aufrufgraphen (1)

...der Rechenvorschriften bzw. Systeme von Rechenvorschriften

add, add', fac, fib, max und mx:

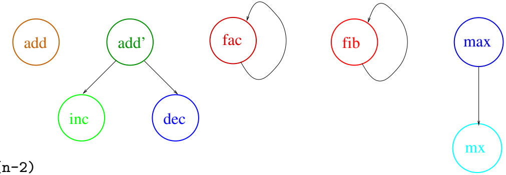
```
add :: Int -> Int -> Int
add m n = (+) m n

add'  :: Int -> Int -> Int
add' m n | n == 0    = m
         | n >0      = add' (inc m) (dec n)
         | otherwise = add' (dec m) (inc n)
 where inc :: Int -> Int
       inc n = n+1
       dec :: Int -> Int
       dec n = n-1

fac :: Integer -> Integer
fac n | n == 0    = 1
      | otherwise = n * fac (n-1)

fib :: Integer -> Integer
fib n | n == 0    = 0
      | n == 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

max :: Int -> Int -> Int -> Int
max p q r
 | (mx p q == p) && (p 'mx' r == p) = p
 | (mx p q == q) && (q 'mx' r == q) = q
 | otherwise                        = r
 where mx :: Int -> Int -> Int
       mx p q | p >= q    = p
              | otherwise = q
```

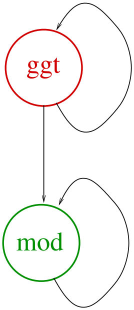


Beispiele: Die Aufrufgraphen (2)

...des Systems hierarchischer Rechenvorschriften der Funktionen ggt und mod:

```
ggt :: Int -> Int -> Int
ggt m n
 | n == 0 = m
 | n > 0  = ggt n (mod m n)

mod :: Int -> Int -> Int
mod m n
 | m < n  = m
 | m >= n = mod (m-n) n
```

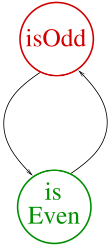


Beispiele: Die Aufrufgraphen (3)

...des Systems wechselweise rekursiver Rechenvorschriften der Funk- tionen `isOdd` und `isEven`:

```
isOdd :: Integer -> Bool
isOdd n
 | n == 0 = False
 | n > 0  = isEven (n-1)

isEven :: Integer -> Bool
isEven n
 | n == 0 = True
 | n > 0  = isOdd (n-1)
```



Beispiele: Die Aufrufgraphen (4)

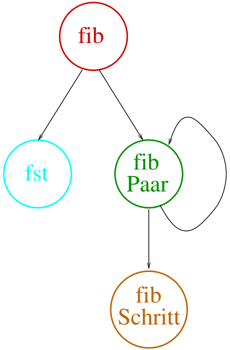
...des Systems hierarchischer Rechenvorschriften der Funktionen `fib`, `fibPaar`, `fibSchritt` und `fst`:

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n =
 | n == 0    = (0,1)
 | otherwise = fibSchritt (fibPaar (n-1))

fib :: Integer -> Integer
fib n = fst (fibPaar n)

fst :: (a,b) -> a
fst (x,y) = x
```



Zur Interpretation von Aufrufgraphen (1)

Aus den Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ablesbar:

- Direkte Rekursivität einer Funktion: "Selbstkringel". (z.B. bei den Aufrufgraphen der Funktionen `fac` und `fib`)
- Wechselweise Rekursivität zweier (oder mehrerer) Funktionen: Kreise (mit mehr als einer Kante) (z.B. bei den Aufrufgraphen der Funktionen `isOdd` und `isEven`)
- Direkte hierarchische Abstützung einer Funktion auf eine andere: Es gibt eine Kante von Knoten f zu Knoten g, aber nicht umgekehrt. (z.B. bei den Aufrufgraphen der Funktionen `max` und `mx`)

Zur Interpretation von Aufrufgraphen (2)

- Indirekte hierarchische Abstützung einer Funktion auf eine andere: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar, aber nicht umgekehrt.
- Wechselweise Abstützung: Knoten g ist von Knoten f direkt oder indirekt über eine Folge von Kanten erreichbar und umgekehrt.
- Unabhängigkeit/Isolation einer Funktion: Knoten f hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.
  (z.B. bei den Aufrufgraphen der Funktionen `add`, `fac` und `fib`)
- ...

Kapitel 7.4

Komplexitätsklassen

Komplexitätsklassen

Komplexitätsklassen und deren Veranschaulichung hier nach

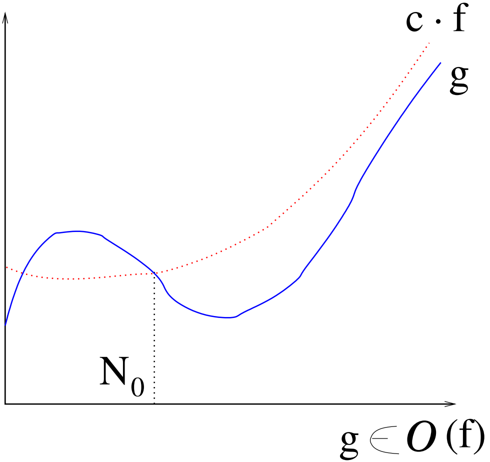
- Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer, Springer-V, 2. Auflage, 2003, Kapitel 11.

$\mathcal{O}$-Notation:

Sei f eine Funktion $f : \alpha \rightarrow IR^+$ von einem gegebenen Datentyp α in die Menge der positiven reellen Zahlen. Dann ist die Klasse $\mathcal{O}(f)$ die Menge aller Funktionen, die "langsamer wachsen" als f:

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Veranschaulichung



Beispiele

...häufig auftretender Kostenfunktionen:

Kürzel	Aufwand	Intuition: *Vertausendfachte Eingabe heißt...*
$\mathcal{O}(c)$	konstant	gleiche Arbeit
$\mathcal{O}(log\ n)$	logarithmisch	nur zehnfache Arbeit
$\mathcal{O}(n)$	linear	auch vertausendfachte Arbeit
$\mathcal{O}(n\ log\ n)$	"$n\ log\ n$"	zehntausendfache Arbeit
$\mathcal{O}(n^2)$	quadratisch	millionenfache Arbeit
$\mathcal{O}(n^3)$	kubisch	milliardenfache Arbeit
$\mathcal{O}(n^c)$	polynomial	gigantisch viel Arbeit (f. großes c)
$\mathcal{O}(2^n)$	exponentiell	hoffnungslos

Veranschaulichung

...was wachsende Größen von Eingaben in realen Zeiten praktisch bedeuten können:

n	linear	quadratisch	kubisch	exponentiell
1	1 μs	1 μs	1 μs	2 μs
10	10 μs	100 μs	1 ms	1 ms
20	20 μs	400 μs	8 ms	1 s
30	30 μs	900 μs	27 ms	18 min
40	40 μs	2 ms	64 ms	13 Tage
50	50 μs	3 ms	125 ms	36 Jahre
60	60 μs	4 ms	216 ms	36 560 Jahre
100	100 μs	10 ms	1 sec	$4 * 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Folgerung

Die vorigen Beispiele und Überlegungen machen deutlich:

- Rekursionsmuster beeinflussen die Effizienz einer Implementierung (siehe die baumartig-rekursive naive Implementierung der Fibonacci-Funktion).
- Die Wahl eines zweckmäßigen und zweckmäßig eingesetzten Rekursionsmusters ist deshalb äußerst wichtig.

Beachte: Nicht das baumartige Rekursionsmuster an sich

- ist ein Problem, sondern sein unzweckmäßiger Einsatz, wenn er wie im Fall der Fibonacci-Funktion zu (unnötigen) Vielfachfachberechnungen von Werten führt!
- Zweckmäßig eingesetzt bietet baumartige Rekursion viele Vorteile, darunter zur Parallelisierung! *Stichwort*: Teile und herrsche (divide and conquer, divide et impera)!

Kapitel 7.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 7 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 4, Rekursion als Entwurfstechnik; Kapitel 9, Laufzeitanalyse von Algorithmen; Kapitel 9.2, Landau-Symbole)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 11, Software-Komplexität)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 5, Rekursion; Kapitel 11, Formalismen 3: Aufwand und Terminierung)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 7 (2)

- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 4.1.3, Induktiv definierte Algorithmen. Türme von Hanoi)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 19, Time and space behaviour)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 20, Time and space behaviour)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 7 (3)

Ingo Wegener. *Komplexität*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 119-144, 2006. (Kapitel 5.1, Größenordnungen und die $\mathcal{O}$-Notation)

Kapitel 8

Auswertung von Ausdrücken

Auswertung

...einfacher und funktionaler Ausdrücke.

Zentral: Die Organisation des Zusammenspiels von

- Expandieren ($\rightsquigarrow$ Funktionsaufrufe)
- Simplifizieren ($\rightsquigarrow$ einfache Ausdrücke)

um einen Ausdruck soweit zu vereinfachen wie möglich.

Kapitel 8.1

Auswertung einfacher Ausdrücke

Auswerten einfacher Ausdrücke

Viele (Simplifikations-) Wege führen zum (selben!) Ziel, hier zum Wert 42:

Weg 1:

```
3 * (9+5) ->> 3 * 14
          ->> 42
```

Weg 2:

```
3 * (9+5) ->> 3*9 + 3*5
          ->> 27 + 3*5
          ->> 27 + 15
          ->> 42
```

Weg 3:

```
3 * (9+5) ->> 3*9 + 3*5
          ->> 3*9 + 15
          ->> 27 + 15
          ->> 42
```

Kapitel 8.2

Auswertung funktionaler Ausdrücke

Auswerten funktionaler Ausdrücke: Bsp. 1

Der Ausdruck `zip [1,3,5] [2,4,6,8,10]` hat den Wert `[(1,2),(3,4),(5,6)]`; seine Semantik ist der Wert `[(1,2),(3,4),(5,6)]`:

```
    zip [1,3,5] [2,4,6,8,10]
->> zip (1:[3,5]) (2:[4,6,8,10]) -- Listenk. sichtbarmachen
->> (1,2) : zip [3,5] [4,6,8,10]   -- Listenk. herausziehen
->> (1,2) : zip (3:[5]) (4:[6,8,10]) -- Listenk. sichtbarm.
->> (1,2) : ((3,4) : zip [5] [6,8,10])     -- L.k. herausz.
->> (1,2) : ((3,4) : zip (5:[]) (6:[8,10]))-- L.k. sichtbarm.
->> (1,2) : ((3,4) : ((5,6) : zip [] [8,10]))-- Lk. herausz.
->> (1,2) : ((3,4) : ((5,6) : []))   -- Ausw. von zip endet
->> (1,2) : ((3,4) : [(5,6)])     -- Syntaxzucker einführen
->> (1,2) : [(3,4),(5,6)]         -- Syntaxzucker einführen
->> [(1,2),(3,4),(5,6)]
```

Auswerten funktionaler Ausdrücke: Bsp. 2 (1)

```
simple x y z ::  Int -> Int -> Int -> Int
simple x y z = (x + z) * (y + z)
```

Weg 1:

```
                          simple 2 3 4
   (Expandieren)    ->> (2 + 4) * (3 + 4)
   (Simplifizieren) ->> 6 * (3 + 4)
               (S)  ->> 6 * 7
               (S)  ->> 42
```

Weg 2:

```
                          simple 2 3 4
               (E)  ->> (2 + 4) * (3 + 4)
               (S)  ->> (2 + 4) * 7
               (S)  ->> 6 * 7
               (S)  ->> 42
```

Weg...

Auswerten funktionaler Ausdrücke: Bsp. 3 (1)

```
fac ::  Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
fac 2
   (Expandieren) ->> if 2 == 0 then 1
                     else (2 * fac (2 - 1))
   (Simplifizieren) ->> 2 * fac (2 - 1)
```

Für die Fortführung der Berechnung

- gibt es jetzt verschiedene Möglichkeiten; wir haben Freiheitsgrade

Zwei dieser Möglichkeiten

- verfolgen wir in der Folge genauer

Auswerten funktionaler Ausdrücke: Bsp. 3 (2)

Variante a)

```
                 2 * fac (2 - 1)
(Simplifizieren) ->> 2 * fac 1
(Expandieren)    ->> 2 * (if 1 == 0 then 1
                              else (1 * fac (1-1)))
                 ->> ... in diesem Stil fortfahren
```

Variante b)

```
                 2 * fac (2 - 1)
(Expandieren)    ->> 2 * (if (2-1) == 0 then 1
                       else ((2-1) * fac ((2-1)-1)))
(Simplifizieren) ->> 2 * ((2-1) * fac ((2-1)-1))
                 ->> ... in diesem Stil fortfahren
```

Auswertung gemäß Variante a)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))

fac 2
   (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
   (S) ->> 2 * fac (2 - 1)
   (S) ->> 2 * fac 1
   (E) ->> 2 * (if 1 == 0 then 1
                 else (1 * fac (1 - 1)))
   (S) ->> 2 * (1 * fac (1 - 1))
   (S) ->> 2 * (1 * fac 0)
   (E) ->> 2 * (1 * (if 0 == 0 then 1
                      else (0 * fac (0 - 1))))
   (S) ->> 2 * (1 * 1)
   (S) ->> 2 * 1
   (S) ->> 2
```

$\rightsquigarrow$ sog. applikative Auswertung.

Auswertung gemäß Variante b)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))

fac 2
   (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
   (S) ->> 2 * fac (2 - 1)
   (E) ->> 2 * (if (2-1) == 0 then 1
                  else ((2-1) * fac ((2-1)-1)))
   (S) ->> 2 * ((2-1) * fac ((2-1)-1))
   (S) ->> 2 * (1 * fac ((2-1)-1))
   (E) ->> 2 * (1 * (if ((2-1)-1) == 0 then 1
                 else ((2-1)-1) * fac (((2-1)-1)-1)))
   (S) ->> 2 * (1 * 1)
   (S) ->> 2 * 1
   (S) ->> 2
```

⇝ sog. normale Auswertung.

Applikative Auswertung des Aufrufs fac 3

```
        fac 3
(E) ->> if 3 == 0 then 1 else (3 * fac (3-1))
(S) ->> if False then 1 else (3 * fac (3-1))
(S) ->> 3 * fac (3-1)
(S) ->> 3 * fac 2
(E) ->> 3 * (if 2 == 0 then 1 else (2 * fac (2-1)))
(S) ->> 3 * (if False then 1 else (2 * fac (2-1)))
(S) ->> 3 * (2 * fac (2-1))
(S) ->> 3 * (2 * fac 1)
(E) ->> 3 * (2 * (if 1 == 0 then 1 else (1 * fac (1-1))))
(S) ->> 3 * (2 * (if False then 1 else (1 * fac (1-1))))
(S) ->> 3 * (2 * (1 * fac (1-1)))
(S) ->> 3 * (2 * (1 * fac 0))
(E) ->> 3 * (2 * (1 * (if 0 == 0 then 1 else (0 * fac (0-1)))))
(S) ->> 3 * (2 * (1 * (if True then 1 else (0 * fac (0-1)))))
(S) ->> 3 * (2 * (1 * (1)))
(S) ->> 3 * (2 * (1 * 1))
(S) ->> 3 * (2 * 1)
(S) ->> 3 * 2
(S) ->> 6
```

Normale Auswertung des Aufrufs fac 3 (1)

```
        fac 3
(E) ->> if 3 == 0 then 1 else (3 * fac (3-1))
(S) ->> if False then 1 else (3 * fac (3-1))
(S) ->> 3 * fac (3-1)
(E) ->> 3 * (if (3-1) == 0 then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (if 2 == 0 then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * (if False then 1 else ((3-1) * fac ((3-1)-1)))
(S) ->> 3 * ((3-1) * fac ((3-1)-1))
(S) ->> 3 * (2 * fac ((3-1)-1))
(E) ->> 3 * (2 * (if ((3-1)-1) == 0 then 1
                    else ((3-1)-1) * fac (((3-1)-1)-1)))
(S) ->> 3 * (2 * (if (2-1) == 0 then 1
                    else ((3-1)-1) * fac (((3-1)-1)-1)))
(S) ->> 3 * (2 * (if 1 == 0 then 1
                    else ((3-1)-1) * fac (((3-1)-1)-1)))
(S) ->> 3 * (2 * (if False then 1
                    else ((3-1)-1) * fac (((3-1)-1)-1)))
```

Normale Auswertung des Aufrufs fac 3 (2)

```
(S) ->> 3 * (2 * ((3-1)-1) * fac (((3-1)-1)-1))
(S) ->> 3 * (2 * (2-1) * fac (((3-1)-1)-1))
(S) ->> 3 * (2 * (1 * fac (((3-1)-1)-1)))
(E) ->> 3 * (2 * (1 *
         (if (((3-1)-1)-1) == 0 then 1
          else ((((3-1)-1)-1) * fac (((((3-1)-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
         (if ((2-1)-1) == 0 then 1
          else ((((3-1)-1)-1) * fac (((((3-1)-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
         (if (1-1) == 0 then 1
          else ((((3-1)-1)-1) * fac (((((3-1)-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
         (if 0 == 0 then 1
          else ((((3-1)-1)-1) * fac (((((3-1)-1)-1)-1))))))
(S) ->> 3 * (2 * (1 *
         (if True then 1
          else ((((3-1)-1)-1) * fac (((((3-1)-1)-1)-1))))))
```

Normale Auswertung des Aufrufs fac 3 (3)

```
(S) ->> 3 * (2 * (1 * (1)))
(S) ->> 3 * (2 * (1 * 1))
(S) ->> 3 * (2 * 1)
(S) ->> 3 * 2
(S) ->> 6
```

Applikative Auswertung des Aufrufs natSum 3

```
          natSum 3
 (E) ->> if 3 == 0 then 0 else (natSum (3-1)) + 3
 (S) ->> if False then 0 else (natSum (3-1)) + 3
 (S) ->> (natSum (3-1)) + 3
 (S) ->> (natSum 2) + 3
 (E) ->> (if 2 == 0 then 0 else (natSum (2-1)) + 2) + 3
 (S) ->> (if False then 0 else (natSum (2-1)) + 2) + 3
 (S) ->> ((natSum (2-1)) + 2) + 3
 (S) ->> ((natSum 1) + 2) + 3
 (E) ->> ((if 1 == 0 then 0 else (natSum (1-1)) + 1) + 2) + 3
 (S) ->> ((if False then 0 else (natSum (1-1)) + 1) + 2) + 3
 (S) ->> (((natSum (1-1)) + 1) + 2) + 3
 (S) ->> (((natSum 0) + 1) + 2) + 3
 (E) ->> (((if 0 == 0 then 0 else (natSum (0-1))) + 1) + 2) + 3
 (S) ->> (((if True then 0 else (natSum (0-1))) + 1) + 2) + 3
 (S) ->> (((0) + 1) + 2) + 3
 (S) ->> ((0 + 1) + 2) + 3
 (S) ->> (1 + 2) + 3
 (S) ->> 3 + 3
 (S) ->> 6
```

Normale Auswertung des Aufrufs natSum 3

```
        natSum 3
(E) ->> if 3 == 0 then 0 else (natSum (3-1)) + 3
(S) ->> if False then 0 else (natSum (3-1)) + 3
(S) ->> (natSum (3-1)) + 3
(E) ->> ...
```

Übungsaufgabe: Vervollständigung der Auswertung.

Hauptresultat

...von Alonzo Church und John Barkley Rosser als Vorgriff auf Kapitel 12.3 und Kapitel 13:

Theorem 8.2.1 (Church/Rosser, 1936)

Jede maximale terminierende Folge von Expansions- und Simplifikationsschritten endet mit demselben Wert.

Kapitel 8.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 8 (1)

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 1, Problem Solving, Programming, and Calculation)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 1, Introduction)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 9, Formalismen 1: Zur Semantik von Funktionen)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 8 (2)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 1, Introducing functional programming)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 1, Introducing functional programming)

Kapitel 9

Programmentwicklung, Programmverstehen

Kapitel 9.1

Programmentwicklung

Systematischer Programmentwurf

Grundsätzlich gilt:

- Das Finden eines algorithmischen Lösungsverfahrens
  - ist ein kreativer Prozess
  - kann (deshalb) nicht vollständig automatisiert werden

Dennoch gibt es

- Vorgehensweisen und Faustregeln

die häufig zum Erfolg führen.

Eine

- systematische Vorgehensweise für die Entwicklung rekursiver Programme

wollen wir in der Folge betrachten.

Systematische Programmentwicklung

...für rekursive Programme in einem 5-schrittigen Prozess.

5-schrittiger Entwurfsprozess (Graham Hutton, 2007)

1. Lege die (Daten-) Typen fest
2. Führe alle relevanten Fälle auf
3. Lege die Lösung für die einfachen (Basis-) Fälle fest
4. Lege die Lösung für die übrigen Fälle fest
5. Verallgemeinere und vereinfache das Lösungsverfahren

Dieses Vorgehen werden wir in der Folge an drei Beispielen demonstrieren.

Aufsummieren einer Liste ganzer Zahlen (1)

- Schritt 1: Lege die (Daten-) Typen fest

```
sum :: [Integer] -> Integer
```

- Schritt 2: Führe alle relevanten Fälle auf

```
sum []     =
sum (n:ns) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
sum []     = 0
sum (n:ns) =
```

Aufsummieren einer Liste ganzer Zahlen (2)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
sum []     = 0
sum (n:ns) = n + sum ns
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsverf.

```
5a) sum :: Num a => [a] -> a

5b) sum = foldr (+) 0
```

Gesamtlösung nach Schritt 5:

```
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

Streichen der ersten n Elemente einer Liste (1)

- Schritt 1: Lege die (Daten-) Typen fest

```
drop :: Int -> [a] -> [a]
```

- Schritt 2: Führe alle relevanten Fälle auf

```
drop 0 []           =
drop 0 (x:xs)       =
drop (n+1) []       =
drop (n+1) (x:xs) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
drop 0 []           = []
drop 0 (x:xs)       = x:xs
drop (n+1) []       = []
drop (n+1) (x:xs) =
```

Streichen der ersten n Elemente einer Liste (2)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
drop 0 []           = []
drop 0 (x:xs)       = x:xs
drop (n+1) []       = []
drop (n+1) (x:xs) = drop n xs
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsv.

```
5a) drop :: Integral b => b -> [a] -> [a]

5b) drop 0 xs             = xs
    drop (n+1) []         = []
    drop (n+1) (x:xs)     = drop n xs

5c) drop 0 xs             = xs
    drop _ []             = []
    drop (n+1) (_:xs)     = drop n xs
```

Streichen der ersten n Elemente einer Liste (3)

Gesamtlösung nach Schritt 5:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs           = xs
drop _ []           = []
drop (n+1) (_:xs) = drop n xs
```

Hinweis:

- Muster der Form `(n+1)` werden von neueren Haskell-Versionen nicht mehr unterstützt. Deshalb:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

Entfernen des letzten Elements einer Liste (1)

...genauer: des letzten Elements einer nichtleeren Liste.

- Schritt 1: Lege die (Daten-) Typen fest

```
rmLast :: [a] -> [a]
```

- Schritt 2: Führe alle relevanten Fälle auf

```
rmLast (x:xs) =
```

- Schritt 3: Lege die Lösung für die Basisfälle fest

```
rmLast (x:xs) | null xs   = []
              | otherwise =
```

Entfernen des letzten Elements einer Liste (2)

- Schritt 4: Lege die Lösung für die übrigen Fälle fest

```
rmLast (x:xs) | null xs   = []
              | otherwise = x : rmLast xs
```

- Schritt 5: Verallgemeinere u. vereinfache das Lösungsverf.

```
5a) rmLast :: [a] -> [a] -- keine Verallg. moegl.
5b) rmLast [_]    = []
    rmLast (x:xs) = x : rmLast xs
```

Gesamtlösung nach Schritt 5:

```
rmLast :: [a] -> [a]
rmLast [_]    = []
rmLast (x:xs) = x : rmLast xs
```

Verfeinerter Entwurfsprozess nach Ramsey (1)

Norman Ramsey (2014) schlägt einen vergleichbaren 7- bzw. 8-schritten Entwurfsprozess vor, der einen Entwurfsprozess von Matthias Felleisen et al. (2001) verfeinert:

1A.&1B. Beschreibe die Daten, die die Funktion benutzt.

2. Beschreibe mithilfe der Signatur, einer Kopfzeile und eine Aufgabenbeschreibung, was die Funktion leistet.

3. Gib Beispiele an, die veranschaulichen und zeigen, was die Funktion leistet.

4. Schreibe ein Skelett (eine Definition mit noch auszufüllenden Lücken) der Funktion (engl. template).

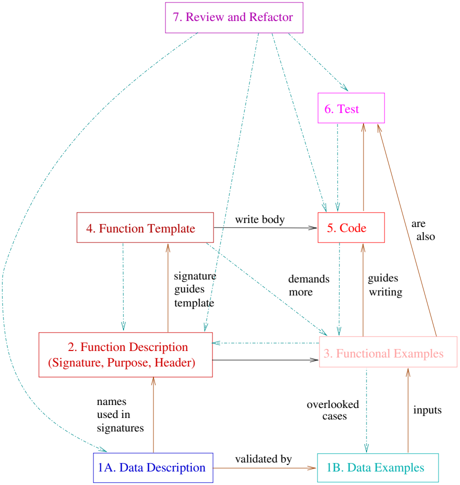
5. Vervollständige das Skelett zu einer vollständigen Funktionsimplementierung (engl. code).

6. Teste die Funktion.

7. Beurteile die Funktion und refaktorisiere sie bei Bedarf.

Verfeinerter Entwurfsprozess nach Ramsey (2)

Graphische Darstellung des Entwurfsprozesses nach Ramsey:



Solid arrows: show initial design
Dotted arrows: show feedback

Norman Ramsey.
On Teaching How to Design Programs.
In Proceedings ICFP 2014, Figure 1, p. 154.

Kapitel 9.2

Programmverstehen

Motivation

Programme häufiger gelesen als geschrieben!

- Eine Binsenweisheit.

Deshalb ist es wichtig, Strategien zu besitzen, die durch geeignete Vorgehensweisen und Fragen an das Programm helfen

- Programme zu lesen und zu verstehen, insbesondere fremde Programme.

Überblick über Vorgehensweisen

Erfolgversprechend:

(1) Lesen des Programmtexts
(2) Nachdenken über das Programm und Ziehen entsprechender Schlussfolgerungen (z.B. Verhaltenshypothesen)

Zur Überprüfung von Verhaltenshypothesen, aber auch zu deren Auffinden kann hilfreich sein:

(3) Gedankliche oder "Papier- und Bleistift"-Programmausführung

Auf einer konzeptuell anderen Ebene hilft das Verständnis des Ressourcenbedarfs, ein Programm zu verstehen:

(4) Analyse des Zeit- und Speicherplatzverhaltens eines Programms

Ein Beispiel

...zur Illustration:

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapWhile f p [] = []                              (mW1)
mapWhile f p (x:xs)
 | p x         = f x : mapWhile f p xs            (mW2)
 | otherwise = []                                 (mW3)
```

(1) Lesen des Programmtexts (1)

Lesen der Funktionssignatur liefert bereits Einsichten in Art und Typ der Argumente und des Resultats. Im Beispiel:

- `mapWhile` erwartet als Argumente
  - eine Funktion `f :: a -> b` eines nicht weiter eingeschränkten Typs `a -> b`
  - eine Eigenschaft von Objekten vom Typ `a`, genauer ein Prädikat (oder Wahrheitsw.funktion) `p :: a -> Bool`
  - eine Liste `l :: [a]` von Elementen vom Typ `a`

  `mapWhile` liefert als Resultat
  - eine Liste `l' :: [b]` von Elementen vom Typ `b`

...zusätzliches Lesen eingestreuter Programmkommentare, auch in Form von Vor- und Nachbedingungen ermöglicht

- weitere und tiefergehende Einsichten.

(1) Lesen des Programmtexts (2)

Lesen der Funktionsdefinition liefert erste weitere Einsichten in Verhalten und Bedeutung des Programms. Im Beispiel:

- Angewendet auf die leere Liste `[]`, ist gemäß (mW1) das Resultat die leere Liste `[]`.
- Angewendet auf eine nichtleere Liste, deren Kopfelement `x` Eigenschaft `p` erfüllt, ist gemäß (mW2) das Element `f x` vom Typ `b` das Kopfelement der Resultatliste, deren Rest sich durch einen rekursiven Aufruf auf die Restliste `xs` ergibt.
- Erfüllt Element `x` die Eigenschaft `p` nicht, bricht gemäß (mW3) die Berechnung ab und liefert als Resultat die leere Liste `[]` zurück.

(2) Nachdenken über das Programm

Nachdenken liefert tiefere Einsichten über Programmverhalten und -bedeutung, auch durch den Beweis von Eigenschaften, die das Programm besitzt. Im Beispiel:

- Für alle Funktionen `f`, Prädikate `p` und endliche Listen `xs` können wir beweisen:

```
mapWhile f p xs
        = map f (takeWhile p xs)           (mW4)
mapWhile f (const True) xs = map f xs      (mW5)
mapWhile id p xs = takeWhile p xs          (mW6)
```

  wobei (mW5) und (mW6) Folgerungen aus (mW4) sind.

(3) Gedankliche, Papier- u. Bleistiftausführung

...hilft, Verhaltenshypothesen zu validieren oder zu generieren durch Berechnung der Funktionswerte für ausgewählte Argumente. Im Beispiel:

```
mapWhile (2+) (>7) [8,12,7,13,16]
 ->> 2+8 : mapWhile (2+) (>7) [12,7,13,16]
                                           wg. (mW2)
 ->> 10 : 2+12 : mapWhile (2+) (>7) [7,13,16]
                                           wg. (mW2)
 ->> 10 : 14 : []                          wg. (mW3)
 ->> [10,14]

mapWhile (2+) (>2) [8,12,7,13,16]
 ->> [10,14,9,15,18]
```

(4) Analyse des Ressourcenverbrauchs

...des Programms liefert

- für das Zeitverhalten: Unter der Annahme, dass `f` und `p` jeweils in konstanter Zeit ausgewertet werden können, ist die Auswertung von `mapWhile` linear in der Länge der Argumentliste, da im schlechtesten Fall die gesamte Liste durchgegangen wird.
- für das Speicherverhalten: Der Platzbedarf ist konstant, da das Kopfelement stets schon “ausgegeben” werden kann, sobald es berechnet ist (siehe unterstrichene Resultatteile):

```
mapWhile (2+) (>7) [8,12,7,13,16]
 ->> 2+8 : mapWhile (2+) (>7) [12,7,13,16]
 ->> 10 : 2+12 : mapWhile (2+) (>7) [7,13,16]
 ->> 10 : 14 : []
 ->> [10,14]
```

Zusammenfassung (1)

Jede der vorgestellten 4 Vorgangsweisen

- bietet einen anderen Zugang zum Verstehen eines Programms.
- liefert für sich einen Mosaikstein zu seinem Verstehen, aus denen sich durch Zusammensetzen ein vollständig(er)es Gesamtbild ergibt.
- kann "von unten nach oben" auch auf Systeme von auf sich wechselweise abstützender Funktionen angewendet werden.
- bietet mit Vorgangsweise (3) der gedanklichen oder Papier- und Bleistiftausführung eines Programms einen stets anwendbaren (Erst-) Zugang zum Erschließen der Programmbedeutung an.

Zusammenfassung (2)

Lesbarkeit und Verständlichkeit eines Programms sollte

- immer schon beim Schreiben des Programms bedacht werden, nicht zuletzt im eigenen Interesse!

Kapitel 9.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (1)

- Matthias Felleisen, Rober B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
- Hugh Glaser, Pieter H. Hartel, Paul W. Garrat. *Programming by Numbers: A Programming Method for Novices*. The Computer Journal 43(4):252-265, 2000.
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 6.6, Advice on Recursion)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 10, Functionally Solving Problems)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (2)

- Norman Ramsey. *On Teaching How to Design Programs*. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014), 153-166, 2014.
- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 4, Induktives Definieren)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 7.4, Finding primitive recursive definitions; Kapitel 14, Designing and writing programs; Kapitel 11, Program development; Anhang D, Understanding programs)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (3)

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 4, Designing and writing programs; Kapitel 7.4, Finding primitive recursive definitions; Kapitel 9.1, Understanding definitions; Kapitel 12.7, Understanding programs)

Teil IV

Funktionale Programmierung

Kapitel 10

Funktionen höherer Ordnung

Kapitel 10.1

Motivation

Funktionen höherer Ordnung

...Bezeichnung für Funktionen, unter deren Argumenten oder Resultaten Funktionen sind.

Damit gilt:

Funktionen höherer Ordnung (oder kurz Funktionale) sind

- spezielle Funktionen.

Beispiele vordefinierter Funktionale in Haskell

Funktionen mit funktionalen Resultaten:

```
(+) :: Num a => a -> (a -> a)
((+) 1) :: Num a => (a -> a)                   -- Inkrementfkt.

splitAt :: Int -> ([a] -> ([a],[a]))
(splitAt 42) :: ([a] -> ([a],[a]))   -- Listenteilungsfkt.
```

Funktionen mit funktionalen Argumenten und Resultaten:

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
(curry binom') :: (Integer -> (Integer -> Integer))
                                                  -- binom-Fkt.

uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
(uncurry binom) :: ((Integer,Integer) -> Integer)
                                                 -- binom'-Fkt.

zipWith :: (a -> (b -> c)) -> ([a] -> ([b] -> [c]))
(zipWith (&&)) :: ([Bool] -> ([Bool] -> [Bool]))
                                          -- "Listen-und"-Fkt.
```

Beispiele selbstdef. Funktionale in Haskell

Funktionen mit funktionalen Argumenten (und nichtfunktionalen Resultaten):

```
f :: ((a -> b),a) -> b
f (g,x) = g x

f (fac,5) = 120 :: Integer
f (reverse,"stressed") = "dessert" :: String
f (concat,[['a','b','c'],['d','e'],[],['f']])
 = ['a','b','c','d','e','f'] :: [Char]
...

h :: ((a -> b -> c),a,b) -> c
h (g,x,y) = g x y

h (binom,49,6) = 13.983.816 :: Integer
h ((++),"Hallo"," Welt!") = "Hallo Welt!" :: String
h (zip,['a','b','c'],[True,False])
 = ([('a',True),('b',False)] :: [(Char,Bool)]
...
```

Bemerkung

...funktionale Programmiersprachen und Programmierung haben eine Präferenz für curryfizierte Funktionsdefinitionen.

Das erklärt die

- Abwesenheit vordefinierter Funktionale mit funktionalen Argumenten ohne funktionale Resultate

in Haskell.

Die Eingangsbeispiele

...zeigen:

Funktionen höherer Ordnung kommen in funktionalen Sprachen

- völlig beiläufig und natürlich daher.

So beiläufig, dass sie in funktionaler Programmierung

- der Regelfall, nicht die Ausnahme

sind.

Funktionen höherer Ordnung auch anderswo

...in der Mathematik:

- Differentialrechnung:

  $\frac{df(x)}{dx}$ $\rightsquigarrow$ `ableitung f x`

  ...Steigung von `f` an der Stelle `x`.

- Integralrechnung:

  $\int_a^b f(x)\,dx$ $\rightsquigarrow$ `integral f a b`

  ...Fläche unterhalb von `f` zwischen `a` und `b`.

- Analysis:

  Theorem. Die Komposition zweier stetiger Funktionen ist wieder eine stetige Funktion, d.h. $(f \circ g)$ mit $(f \circ g)(x) = f(g(x))$ ist stetig, wenn $f, g : \mathbb{R} \rightarrow \mathbb{R}$ stetig sind.

Funktionen höherer Ordnung auch anderswo

...Informatik:

- Semantik von Programmiersprachen:

  Die Bedeutung der while-Schleife im denotationellen Stil

  $$[\![\, \texttt{while b do } \pi \texttt{ od} \,]\!]_{ds} : \Sigma \to \Sigma$$

  ...festgelegt als kleinster Fixpunkt einer Funktion höherer Ordnung auf der Menge der Zustandstransformationen mit

  - V: Menge der Programmvariablen.
  - D: Datenbereich.
  - $\Sigma =_{df} \{ \sigma \mid \sigma : V \to D \}$: Menge der Zustände.
  - $[\Sigma \to \Sigma] =_{df} \{ zt \mid zt : \Sigma \to \Sigma \}$: Menge der Zustandstransformationen.

  (Siehe z.B. VU 185.278 Theoretische Informatik und Logik)

Funktionen höherer Ordnung

...können dennoch überraschen:

"The functions I grew up with, such as the sine, the cosine, the square root, and the logarithm were almost exclusively real functions of a real argument. [...] I was really ill-equipped to appreciate functional programming when I encountered it: I was, for instance, totally baffled by the shocking suggestion that the value of a function could be another function."(∗)

Edsger W. Dijkstra
(11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

(∗) Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas, Austin, 1995.

Mit Funktionen höherer Ordnung

...machen wir den Schritt von applikativer zu funktionaler Programmierung!

Frei nach Hegel:

Der Mensch
wird erst durch Arbeit
zum Menschen.

Georg W.F. Hegel
(27.08.1770-14.11.1831)

...auf den Punkt gebracht:

Die funktionale Programmierung
wird erst durch Funktionen höherer Ordnung
zu funktionaler Programmierung.

Mit Fug und Recht

...die vollumfängliche Integration von Funktionen höherer Ordnung als erstrangige Elemente (engl. first-class citizens)

- ist charakteristisch und kennzeichnend für funktionale Programmierung.
- hebt funktionale Programmierung von anderen Programmierparadigmen ab.
- ist wesentliches sprachliches Mittel funktionaler Sprachen für extrem ausdruckskräftige, elegante und flexible Programmiermethoden, insbesondere zur Unterstützung von Wiederverwendung.

Kapitel 10.2

Funktionale Abstraktion

Abstraktionsprinzipien

Kennzeichnendes Strukturierungsprinzip für

- Prozedurale Sprachen: Prozedurale Abstraktion
- Funktionale Sprachen: Funktionale Abstraktion
  - 1-ter Stufe: Funktionen

    $\rightsquigarrow$ Operanden werden zu elementartypigen Parametern von Funktionen (funktionales Analogon zu prozeduraler Abstraktion).
  - Höherer Stufe: Funktionen höherer Ordnung

    $\rightsquigarrow$ Verknüpfungsvorschriften werden zu funktionalen Parametern von Funktionen höherer Ordnung.

Funktionale Abstraktion 1-ter Stufe (1)

Idee: Operanden werden zu Parametern von Funktionen.

Beispiel: Sind viele strukturell gleiche Ausdrücke auszuwerten wie

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

...führe eine funktionale Abstraktion durch, d.h. schreibe eine Funktion, die die Operanden des Ausdrucksmusters als Parameter erhält:

```
f :: (Int,Int,Int) -> Int
f (a,b,c) = (a * b + c) * (b + a * c)
```

und mit den ursprünglichen Ausdrucksoperanden(werten) aufgerufen wird.

Funktionale Abstraktion 1-ter Stufe (2)

Beispiel (fgs.): Die Funktion `f` erlaubt uns die Berechnungsvorschrift des Ausdrucksmusters `(a * b + c) * (b + a * c)` wiederzuverwenden:

```
f (5,37,13) ->> 20.196
f (15,7,12) ->> 21.879
f (25,3,10) ->> 21.930
...
```

Gewinn: Wiederverwendung durch funktionale Abstraktion.

Funktionale Abstraktion höherer Stufe (1)

Idee: Verknüpfungsvorschriften werden zu funktionalen Parametern einer Funktion höherer Ordnung.

Beispiel: (siehe Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach, Addison-Wesley, 1999, S. 7f.):

- Fakultätsfunktion:

```
fac n | n==0   = 1
      | n>0    = n * fac (n-1)
```

- Summe der n ersten natürlichen Zahlen:

```
natSum n | n==0  = 0
         | n>0   = n + natSum (n-1)
```

- Summe der n ersten natürlichen Quadratzahlen:

```
natQuSum n | n==0  = 0
           | n>0   = n*n + natQuSum (n-1)
```

Funktionale Abstraktion höherer Stufe (2)

Beobachtung:

- Die Definitionen von `fac`, `natSum` und `natQuSum` folgen demselben Rekursionsschema und der strukturell selben Verknüpfungsvorschrift ihrer Argumente.

Dieses gemeinsame Rekursionsschema und die Verknüpfungsvorschrift sind gekennzeichnet durch die Festlegung von im

- Basisfall: eines Basiswerts.
- Rekursionsfall: einer Verknüpfungsvorschrift des Argumentwerts `n` und des Funktionswerts für `(n-1)`.

Funktionale Abstraktion höherer Stufe (3)

Diese Gemeinsamkeit legt es nahe

- Rekursionsschema
- Verknüpfungsvorschrift
- Basiswert

herauszuziehen, zu abstrahieren; eine Abstraktion höherer Stufe.

Das ergibt:

```
rekSchema :: Int -> (Int -> Int -> Int) -> Int -> Int
rekSchema basiswert verknuepfe n
 | n==0 = basiswert
 | n>0  = verknuepfe n (rekSchema basiswert verknuepfe (n-1))
```

Funktionale Abstraktion höherer Stufe (4)

...diese funktionale Abstraktion höherer Stufe erlaubt nun, die Implementierungen von

- `fac`, `natSum` und `natQuSum`

zu ersetzen durch entsprechende Aufrufe der

- Funktion höherer Ordnung `rekSchema`

der die einzelnen Verknüpfungsvorschriften von `fac`, `natSum` und `natQuSum` über den funktionalen Parameter `verknuepfe` übergeben werden.

Funktionale Abstraktion höherer Stufe (5)

Redefinition der Funktionen mittels `rekSchema`:

```
fac      = rekSchema 1 (*)
natSum   = rekSchema 0 (+)
natQuSum = rekSchema 0 (\x y -> x*x + y)
```

...alternativ argumentbehaftet:

```
fac n      = rekSchema 1 (*) n
natSum n   = rekSchema 0 (+) n
natQuSum n = rekSchema 0 (\x y -> x*x + y) n
```

Gewinn: Wiederverwendung des gemeinsamen Strukturmusters der Funktionen `fac`, `natSum` und `natQuSum` durch

- funktionale Abstraktion höherer Stufe.

Zusammenfassung d. `rekSchema`-Beispiels (1)

...die Signatur zeigt, dass `rekSchema` eine Funktion höherer Ordnung ist, die als ein Argument eine Funktion erwartet:

```
rekSchema :: Int -> (Int -> Int -> Int) -> Int -> Int
```

Beachte: Streng genommen, ist `rekSchema` eine einstellige Funktion, die aufgerufen mit einem ganzzahligen Argument `z` eine Funktion höherer Ordnung als Resultat liefert, nämlich den Wert des Funktionsterms `(rekSchema z)` vom Typ

```
(rekSchema z) :: (Int -> Int -> Int) -> Int -> Int
```

Die uncurryfizierte Version von `rekSchema` bzw. mit getauschter Argumentfolge macht deutlicher, dass das Rekursionsschema (u.a.) eine Funktion als Argument erwartet:

```
rekSchema'  :: (Int,(Int -> Int -> Int),Int) -> Int
rekSchema'' :: (Int -> Int -> Int) -> Int -> Int -> Int
```

Zusammenfassung d. rekSchema-Beispiels (2)

Für die Anwendungsbeispiele von `rekSchema` gilt:

	Basisfallwert	Verknüpfungsvorschrift
`fac`	1	`(*)`
`natSum`	0	`(+)`
`natQuSum`	0	`\x y -> x*x + y`

Übungsaufgabe 10.2.1

Ergänze die Deklarationen von

```
rekSchema'  :: (Int,(Int -> Int -> Int),Int) -> Int
rekSchema'' :: (Int -> Int -> Int) -> Int -> Int -> Int
```

zu vollständigen Implementierungen und teste sie mit geeigneten Beispielen.

Zurück zum u. weiter mit d. Eingangsbsp. (1)

- Funktionale Abstraktion 1. Stufe führt von Ausdrücken
  `(5*37+13)*(37+5*13), (15*7+12)*(7+15*12),...`
  zu Funktionen:

```
f:: (Int,Int,Int) -> Int
f (a,b,c) = (a * b + c) * (b + a * c)
```

  Aufrufbeispiele:

```
f (5,37,13) ->> 20.196
f (15,7,12) ->> 21.879
 ...
```

- Funktionale Abstraktion höherer Stufe führt von Funktionen zu Funktionen höherer Ordnung:

```
fho :: (((Int,Int,Int)->Int),Int,Int,Int) -> Int
fho (g,a,b,c) = g (a,b,c)
```

  Aufrufbeispiele:

```
fho (f,5,37,13) ->> 20.196
fho (f,15,7,12) ->> 21.879
...
```

Zurück zum u. weiter mit d. Eingangsbsp. (2)

...zusätzlich zur

- freien Wahl der elementaren Argumentwerte

(wie f) erlaubt die Funktion höherer Ordnung fho auch die

- freie Wahl der Vorschrift sie zu verknüpfen.

Beispiele:

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)

g :: Int -> Int -> Int -> Int
g a b c = a^b `div` c

h :: Int -> Int -> Int -> Int
h a b c = if (a `mod` 2 == 0) then b else c
```

Zurück zum u. weiter mit d. Eingangsbsp. (3)

Aufrufbeispiele:

```
fho (f,2,3,5) ->> f 2 3 5
              ->> (2*3+5)*(3+2*5)
              ->> (6+5)*(3+10)
              ->> 11*13
              ->> 143

fho (g,2,3,5) ->> g 2 3 5
              ->> 2^3 `div` 5
              ->> 8 `div` 5
              ->> 1

fho (h,2,3,5) ->> h 2 3 5
              ->> if (2 `mod` 2 == 0) then 3 else 5
              ->> if (0 == 0) then 3 else 5
              ->> if True then 3 else 5
              ->> 3
```

Zusammenfassung

...Gewinn durch funktionale Abstraktion:

- Wiederverwendung und dadurch kürzerer, verlässlicherer, wartungsfreundlicherer Code.

Zwingend erforderlich für erfolgreiches Gelingen:

- Funktionen höherer Ordnung (oder kurz Funktionale).

Zum Abschluss:

- Allgemeinster Typ des Funktionals `rekSchema` ist (siehe Kap. 11 und Kap. 14):

```
rekSchema :: (Num a, Ord a) =>
                    b -> (a -> b -> b) -> a -> b
```

Kapitel 10.3

Funktionen als Argument

Funktionen als Argument: 1-tes Beispiel (1)

Betrachte die spezialisierten Vergleichsfunktionen `min`, `max`:

```
min :: Ord a => a -> a -> a
min x y
  | x < y      = x
  | otherwise = y

max :: Ord a => a -> a -> a
max x y
  | x > y      = x
  | otherwise = y
```

...Abstraktion höherer Stufe und herausziehen der Vergleichsoperation erlaubt die Vergleichsfunktionen zu generalisieren...

Funktionen als Argument: 1-tes Beispiel (2)

...zu einer mit einer Wahrheitswertfunktion parametrisierten Funktion höherer Ordnung `extreme`, die `min`, `max` zu redefinieren erlaubt:

```
extreme :: Ord a => (a -> a -> Bool) -> a -> a -> a
extreme wwf m n
 | wwf m n   = m
 | otherwise = n
```

```
min = extreme (<)                        -- argumentfrei
max = extreme (>)
```

```
min x y = extreme (<) x y           -- argumentbehaftet
max x y = extreme (>) x y
```

...oder auch gänzlich (durch Aufrufe v. `extreme`) zu ersetzen:

```
max 17 4 ->> extreme (>) 17 4 ->> 17
min 17 4 ->> extreme (<) 17 4 ->> 4
```

Funktionen als Argument: 2-tes Beispiel (1)

Betrachte die Funktion `zip`:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  _      _     = []
```

...und die Funktion höherer Ordnung `zipWith`:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f  _      _     = []
```

`zipWith` erlaubt `zip` zu implementieren (und zu ersetzen):

```
zip :: [a] -> [b] -> [(a,b)]
zip xs ys  = zipWith v xs ys
              where v :: a -> b -> (a,b)
                    v = (,)  -- (,) Paarbildungsop.
-- v x y = (x,y) gleichbedeutend zu: v x y = (,) x y
```

Funktionen als Argument: 2-tes Beispiel (2)

...aufgrund der Parametrisierung leistet `zipWith` mehr als `zip` zu implementieren (und ist in diesem Sinn genereller).

Betrachte dazu etwa folgende Beispiele:

```
f :: a -> b -> (a,b)
f x y = (x,y)

g :: a -> a -> [a]
g x y = [x,y]

h :: Num a => a -> a -> a
h x y = x+y

k :: Ord a => a -> a -> Bool
k x y = x > y

zipWith f ['a','b'] [1,2,3] ->> [('a',1),('b',2)]
zipWith g [1,2,3] [5,6,7,8] ->> [[1,5],[2,6],[3,7]]
zipWith h [1,2,3] [10,20,30,40] ->> [11,22,33]
zipWith k [10,20,30] [5,15,35,85] ->> [True,True,False]
```

Funktionen als Argument: 3-tes Beispiel

Transformation der Marken eines benannten Baums bzw. Herausfiltern der Marken mit einer bestimmten Eigenschaft:

```
data Baum a = Leer | Wurzel a (Baum a) (Baum a)

map_Baum :: (a -> a) -> Baum a -> Baum a
map_Baum _ Leer = Leer
map_Baum tf (Wurzel marke ltb rtb) =
 Wurzel (tf marke) (map_Baum tf ltb) (map_Baum tf rtb)

filter_Baum :: (a -> Bool) -> Baum a -> [a]
filter_Baum _ Leer = []
filter_Baum wwf (Wurzel marke ltb rtb)
 | wwf marke = marke : ((filter_Baum wwf ltb)
                ++ (filter_Baum wwf rtb))
 | otherwise = (filter_Baum wwf ltb)
                ++ (filter_Baum wwf rtb)
```

...mithilfe zweier Funktionen höherer Ordnung, die parametrisiert sind in Transformationsfunktion bzw. Wahrheitswertfunktion.

Zusammenfassung

Funktionen als Argument

- erhöhen die Ausdruckskraft.
- unterstützen Wiederverwendung.
- sind charakteristisch für funktionale Programmierung.

Kapitel 10.4

Funktionen als Resultat

Funktionen als Resultat

...der Regelfall, nicht die Ausnahme in funktionalen Sprachen.

Betrachte zum Beispiel:

```
(+) :: Num a => a -> a -> a
binom :: Integer -> Integer -> Integer
rekSchema :: (Num a, Ord a) =>
                 b -> (a -> b -> b) -> a -> b
...
```

Klammerung hebt die funktionalen Resultate besonders hervor:

```
(+) :: Num a => a -> (a -> a)
binom :: Integer -> (Integer -> Integer)
rekSchema :: (Num a, Ord a) =>
                 b -> ((a -> b -> b) -> (a -> b))
...
```

Funktionen als Resultat: Weitere Beispiele

Wiederholtes Anwenden:

```
iterate :: Int -> (a -> a) -> (a -> a)
iterate n f
 | n > 0     = f . iterate (n-1) f  -- (.) Funktions-
                                    -- komposition
 | otherwise = id
 where
  id :: a -> a            -- Typvariable und Parameter
  id a = a                -- dürfen gleichbenannt sein.

(iterate 3 square) 2
  ->> (square . square . square . id) 2 ->> 256
```

Vertauschen von Argumenten:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x

flip (-) 3 5  ->> (-) 5 3 ->> 2
(flip . flip) ->> id
```

1) Funktionen als Resultat

...durch explizites Ausprogrammieren:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
flip :: (a -> b -> c) -> (b -> a -> c)
iterate :: Int -> (a -> a) -> (a -> a)
extreme :: Ord a => (a -> a -> Bool) -> (a -> a -> a)

addFuns :: Num a => (a -> a) -> (a -> a) -> (a -> a)
addFuns f g = \x -> f x + g x

funny :: (Ord a, Num a) => (a -> a) -> (a -> a)
                                   -> (a -> a)
funny f g = \x -> if x >= 0 then (g . f) x
                  else addFuns f g (x+1)
```

Vergleiche die Definitionen von `addFuns` und `addFuns'`:

```
addFuns' :: Num a => (a -> a) -> (a -> a) -> (a -> a)
addFuns' f g x = f x + g x
```

2) Funktionen als Resultat

...durch partielle Auswertung curryfizierter Funktionen:

```
((+) 1) :: Num a => a -> a
(binom 45) :: Integer -> Integer
(rekSchema 0) :: (Num a, Ord a, Num b) =>
                    (a -> b -> b) -> (a -> b)
(rekSchema 0 (+)) :: (Num a, Ord a) => a -> b
(extreme (<)) :: Ord a => a -> a -> a
(extreme (<) 5) :: (Num a, Ord a) => a -> a
(iterate 5) :: (a -> a) -> (a -> a)
(iterate 5 fac) :: Integer -> Integer
(flip (-)) :: Num a => a -> a -> a
addFuns fac fac :: Integer -> Integer
funny fac fac :: Integer -> Integer
...
```

3) Funktionen als Resultat

...durch Operatorabschnitte (als Spezialfall partieller Auswertung für binäre Operatoren und Funktionen):

```
(+1) :: Num a => a -> a            -- Inkrementieren
(-1) :: Num a => a -> a            -- Dekrementieren
(2*) :: Num a => a -> a                -- Verdoppeln
(<2) :: (Num a, Ord a) => a -> a    -- Kleiner 2?
(== True) :: Bool -> Bool                -- Wahr?
(True &&) :: Bool -> Bool                -- Wahr?
(42:) :: Num a => a -> [a] -> [a]
                          -- 42 als neuer Listenkopf
(45 'binom') :: Integer -> Integer    -- 45 über k
(47 '(extreme (<))') :: (Num a, Ord a) => a -> a
                             -- Minimum aus 47 und x
...
```

4) Funktionen als Resultat

...durch Bildung konstanter Funktionen (engl. λ-Lifting):

```
lifting :: a -> (b -> a)
lifting c = \x -> c
```

Anwendungsbeispiele:

```
lifting 42 "Aller Fragen Antwort"      ->> 42
lifting iterate flip                   ->> iterate
lifting (iterate (+) 3 (\x->x*x)) 42 2 ->> 256
```

5) Funktionen als Resultat

...durch Komposition von Funktionen:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Wichtige Eigenschaft von (.): Assoziativität

```
(f . (g . h)) = ((f . g) . h) = (f . g . h)
```

Beachte: Funktionskomposition und Funktionsapplikation sind grundverschieden und auseinanderzuhalten:

- Komposition: `(f . g) x = f (g x) = f(g(x))`
- Applikation: `(f g) x = (f g) x = (f(g))(x)`

Übungsaufgabe 10.4.1

Überprüfe und teste die unterschiedliche Wirkung von Komposition und Applikation

- Komposition: `(f . g) x = f (g x) = f(g(x))`
- Applikation: `(f g) x = (f g) x = (f(g))(x)`

anhand geeigneter Beispiele für `f`, `g` und `x`.

Beachte, dass sich eine Funktion `f`, die sich mit einer Funktion `g` komponieren lässt, nicht notwendig auf `g` applizieren lässt und umgekehrt.

Gibt es Beispiele für `f`, `g` und `x`, so dass sowohl

```
(f . g) x
```

als auch

```
(f  g) x
```

gültige Ausdrücke sind?

Funktionskomposition: Anwendungsbsp. (1)

Das 4-te Element einer Liste:

```
gib_4tes_Element :: [a] -> a
gib_4tes_Element = head . dreimal_rest

dreimal_rest :: [a] -> [a]
dreimal_rest = tail . tail . tail
```

Das n-te Element einer Liste:

```
gib_ntes_Element :: Int -> [a] -> a
gib_ntes_Element n = (head . (iterate (n-1) tail))
```

...Funktionskomposition ermöglicht Funktionsdefinitionen auf dem (Abstraktions-) Niveau von Funktionen statt von (elementaren) Werten.

Funktionskomposition: Anwendungsbsp. (2)

...Definitionen auf Funktionsniveau sind kürzer und meist einfacher zu verstehen als ihre argumentbehafteten Gegenstücke.

Zum Vergleich argumentfreie und argumentbehaftete Implementierungen:

```
gib_4tes_Element :: [a] -> a
gib_4tes_Element = head . dreimal_rest

gib_4tes_Element ls = (head . dreimal_rest) ls
gib_4tes_Element ls = head (dreimal_rest ls)

gib_ntes_Element :: Int -> [a] -> a
gib_ntes_Element = head . (iterate tail)

gib_ntes_Element n = (head . (iterate tail)) n
gib_ntes_Element n lst
 = (head . (iterate tail) n) lst
```

Zusammenfassung

Funktionen als Resultat

- erhöhen die Ausdruckskraft.
- unterstützen Wiederverwendung.
- sind kennzeichnend für funktionale Programmierung.

Insgesamt: Funktionen gleichberechtigt zu elementaren Werten als Argument und Resultat von Funktionen zuzulassen

- ist maßgeblich für Ausdruckskraft, Eleganz und Prägnanz funktionaler Programmierung.
- zeichnet funktionale Programmierung signifikant vor anderen Programmierparadigmen aus.

Kapitel 10.5

Funktionale auf Listen

Funktionale auf Listen

...ein wichtiger Spezialfall.

Vordefinierte Listenfunktionale für häufige Problemstellungen in Haskell (und anderen funktionalen Programmiersprachen):

- Transformieren aller Listenelemente mittels einer Abbildungsvorschrift:
  ```
  map :: (a -> b) -> [a] -> [b]
  ```
- Herausfiltern aller Listenelemente mit einer bestimmten Eigenschaft:
  ```
  filter :: (a -> Bool) -> [a] -> [a]
  ```
- Aggregieren aller Listenelemente mittels einer Verknüpfungsoperation:
  ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldl :: (a -> b -> a) -> a -> [b] -> a
  ```
- ...

Transformieren: Das Funktional map (1)

Signatur:

```
map :: (a -> b) -> [a] -> [b]
```

Implementierung mittels (expliziter) Rekursion:

```
map f []     = []
map f (l:ls) = (f l) : map f ls
```

Implementierung mittels Listenkomprehension:

```
map f ls = [ f l | l <- ls ]
```

Anwendungsbeispiele:

```
map square [2,4..10] ->> [4,16,36,64,100]
map length ["abc","abcde","ab"] ->> [3,5,2]
map (>0) [4,(-3),2,(-1),0,2]
    ->> [True,False,True,False,False,True]
```

Transformieren: Das Funktional map (2)

Anwendungsbeispiele (fgs.):

```
map (*) [2,4..10] ->> [(2*),(4*),(6*),(8*),(10*)]
                        :: [Integer -> Integer]
map (-) [2,4..10] ->> [(2-),(4-),(6-),(8-),(10-)]
                        :: [Integer -> Integer]
map (>) [2,4..10] ->> [(2>),(4>),(6>),(8>),(10>)]
                        :: [Integer -> Bool]

[ f 10 | f <- map (*) [2,4..10] ]
   ->> [20,40,60,80,100]
[ f 100 | f <- map (-) [2,4..10] ]
   ->> [-98,-96,-94,-92,-90]
[ f 5 | f <- map (>) [2,4..10] ]
   ->> [False,False,True,True,True]
```

Transformieren: Das Funktional map (3)

Einige Eigenschaften von map:

- Für alle Abbildungsvorschriften f, g gilt:

```
map (\x -> x)      =  \x -> x
map (f . g)        =  map f . map g
map f . tail       =  tail . map f
map f . reverse    = reverse . map f
map f . concat     = concat . map (map f)
map f (xs ++ ys) = map f xs ++ map f ys
```

- Für strikte Abbildungsvorschriften f gilt:

```
f . head  =  head . (map f)
```

Filtern: Das Funktional filter

Signatur:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Implementierung mittels (expliziter) Rekursion:

```
filter p []       = []
filter p (l:ls)
 | p l            = l : filter p ls
 | otherwise      = filter p ls
```

Implementierung mittels Listenkomprehension:

```
filter p ls = [ l | l <- ls, p l ]
```

Anwendungsbeispiel:

```
filter istZweierPotenz [2,4..100] ->> [2,4,8,16,32,64]
```

Aggregieren, Falten von Listen: Motivation

Aufgabe: Berechne die Summe der Elemente einer Liste:

```
sum [1,2,3,4,5] ->> 15
```

Zwei Rechenweisen sind naheliegend zur Aufgabenlösung:

- Summieren (bzw. aggregieren, falten) von rechts:

```
(1+(2+(3+(4+5)))) ->> (1+(2+(3+9)))
                  ->> (1+(2+12))
                  ->> (1+14) ->> 15
```

- Summieren (bzw. aggregieren, falten) von links:

```
((((1+2)+3)+4)+5) ->> (((3+3)+4)+5)
                  ->> ((6+4)+5)
                  ->> (10+5) ->> 15
```

...die Funktionale `foldr` und `foldl` systematisieren diese Rechenweisen.

Aggregieren: Das Funktional foldr (1)

Signatur (`foldr`: falten, zusammenfassen von rechts):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Implementierung mittels (expliziter) Rekursion:

```
foldr f e []     = e
foldr f e (l:ls) = f l (foldr f e ls)
```

Es bedeuten:

- `f`: Faltungsvorschrift.
- `e`: Auffangwert, Vorgabewert für leere Argumentliste.
- `[]`, `(l:ls)`: Liste zu aggregierender Werte.

Aggregieren: Das Funktional foldr (2)

Anwendungsbeispiele:

```
foldr (+) 0 [2,4..10]
 ->> ((+) 2 ((+) 4 ((+) 6 ((+) 8 ((+) 10 0)))))
 ->> (2 + (4 + (6 + (8 + (10 + 0))))) ->> 30
foldr (+) 0 [] ->> 0

foldr (*) 1 [2,4..10]
 ->> ((*) 2 ((*) 4 ((*) 6 ((*) 8 ((*) 10 1)))))
 ->> (2 * (4 * (6 * (8 * (10 * 1))))) ->> 3.840
foldr (*) 1 [] ->> 1

foldr (||) False [True,False,False]
->> ((||) True ((||) False ((||) False False)))
->> (True || (False || (False || False))) ->> True
foldr (||) False [] ->> False
```

Aggregieren: Das Funktional foldr (3)

Anwendungsbeispiele (fgs.): Definition einiger Standardfunktionen in Haskell mittels foldr:

```
sum :: Num a => [a] -> a
sum ns = foldr (+) 0 ns

prod :: Num a => [a] -> a
prod ns = foldr (*) 1 ns

and :: [Bool] -> Bool
and bs = foldr (&&) True bs

or :: [Bool] -> Bool
or bs = foldr (||) False bs

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

Aggregieren: Das Funktional `foldl` (1)

Signatur (`foldl`: falten, zusammenfassen von links):

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Implementierung mittels (expliziter) Rekursion:

```
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls
```

Es bedeuten:

- `f`: Faltungsvorschrift.
- `e`: Auffangwert, Vorgabewert für leere Argumentliste.
- `[]`, `(l:ls)`: Liste zu aggregierender Werte.

Aggregieren: Das Funktional `foldl` (2)

Anwendungsbeispiele:

```
foldl (+) 0 [2,4..10]
 ->> ((+) ((+) ((+) ((+) ((+) 0 2) 4) 6) 8) 10)
 ->> (((((0 + 2) + 4) + 6) + 8) + 10) ->> 30
foldl (+) 0 [] ->> 0

foldl (*) 1 [2,4..10]
 ->> ((*) ((*) ((*) ((*) ((*) 1 2) 4) 6) 8) 10)
 ->> (((((1 * 2) * 4) * 6) * 8) * 10) ->> 3.840
foldl (*) 1 [] ->> 1

foldl (||) False [True,False,False]
 ->> ((||) ((||) ((||) False True) False) False)
 ->> (((False || True) || False) || False) ->> True
foldl (||) False [] ->> False
```

Aggregieren: Das Funktional `foldl` (3)

Anwendungsbeispiele (fgs.): Alternative Definitionen einiger Standardfunktionen in Haskell mittels `foldl`:

```
sum :: Num a => [a] -> a
sum ns = foldl (+) 0 ns
prod :: Num a => [a] -> a
prod ns = foldl (*) 1 ns

and :: [Bool] -> Bool
and bs = foldl (&&) True bs
or :: [Bool] -> Bool
or bs = foldl (||) False bs

concat :: [[a]] -> [a]
concat xss = foldl (++) [] xss
```

foldr, foldl im Vergleich

foldr: Falten, zusammenfassen von rechts:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (l:ls) = f l (foldr f e ls)

foldr f e [a1,a2,...,an]
->> a1 'f' (a2 'f'...'f' (an-1 'f' (an 'f' e))...)
```

foldl: Falten, zusammenfassen von links:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls

foldl f e [b1,b2,...,bn]
->> (...((e 'f' b1) 'f' b2) 'f'...'f' bn-1) 'f' bn
```

Warum zwei Faltungsfunktionale?

...die Funktionale foldr und foldl unterscheiden sich in

- Anwendbarkeit
- Effizienz

abhängig vom Anwendungskontext.

Zur Illustration betrachten wir die Implementierungen der Funktionen reverse und concat aus dem Präludium:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
 where flip :: (a -> b -> c) -> (b -> a -> c)
       flip f x y = f y x

concat :: [[a]] -> [a]
concat = foldr (++) []
```

Zur Anwendbarkeit von foldl, foldr

Die Implementierung von reverse aus dem Präludium:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
 where flip :: (a -> b -> c) -> (b -> a -> c)
       flip f x y = f y x
```

...leistet die gewünschte Listenumkehrung; sie hat dieselbe Bedeutung wie folgende rekursive Implementierung:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (l:ls) = (reverse ls) ++ [l]

reverse []      ->> []
reverse [1,2,3] ->> [3,2,1]
```

Zur Wirkung von foldl

...im Zusammenspiel mit der Funktion reverse:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
 where flip :: (a -> b -> c) -> (b -> a -> c)
       flip f x y = f y x
```

...für die Argumentlisten [] und [1,2,3]:

```
reverse [] ->> foldl (flip (:)) [] [] ->> []

reverse [1,2,3]
 ->> foldl (flip (:)) [] [1,2,3]
 ->> ((flip (:)) ((flip (:)) ((flip (:)) [] 1) 2) 3)
 ->> ((([] 'flip (:)' 1) 'flip (:)' 2) 'flip (:)' 3)
 ->> (((1 : []) 'flip (:)' 2) 'flip (:)' 3)
 ->> ((2 : (1 : [])) 'flip (:)' 3)
 ->> (3 : (2 : (1 : [])))
 ->> [3,2,1]
```

Zur Wirkung von foldr

..im Zusammenspiel mit der Funktion reverse:

```
rev_untauglich :: [a] -> [a]
rev_untauglich = foldr (flip (:)) []
 where flip :: (a -> b -> c) -> (b -> a -> c)
       flip f x y = f y x
```

...für die Argumentlisten [] und [1,2,3]:

```
rev_untauglich [] ->> foldr (flip (:)) [] [] ->> []

rev_untauglich [1,2,3]
 ->> foldr (flip (:)) [] [1,2,3]
 ->> ((flip (:)) 1 ((flip (:)) 2 ((flip (:)) 3 [])))
 ->> (1 'flip (:)' (2 'flip (:)' (3 'flip (:)' [])))
 ->> (1 'flip (:)' (2 'flip (:)' (3 'flip (:)' [])))
 ->> (1 'flip (:)' (2 'flip (:)' ([] : 3)))
 ->> Typunverträglichkeit d. Operanden im Term ([] : 3).
     Auswertungsversuch von rev_untauglich scheitert!
```

Übungsaufgabe 10.5.1: `(:)` statt `(flip (:))`

Vollziehe ebenfalls mit Papier und Bleistift nach (z.B. für die Argumentliste `[1,2,3]`), dass sich die Faltungsfunktionale `foldl` und `foldr` auch bezüglich der Faltungsoperation `(:)` unterschiedlich verhalten. Zeige dazu, dass folgender Versuch

- untauglich ist; Auswertungsversuche für nichtleere Listen scheitern aufgrund von Operandenunverträglichkeiten:

```
rev_untauglich' :: [a] -> [a]
rev_untauglich' = foldl (:) []
```

- die Identität auf Listen liefert:

```
rev_id :: [a] -> [a]
rev_id = foldr (:) []
```

d.h. für alle Listen `xs` gilt: `rev_id xs ->> xs`

Zur Effizienz von concat, slow_concat (1)

Vergleiche die Effizienz und Performanz von:

- concat wie im Präludium mittels foldr definiert:

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

...mit derjenigen von:

- slow_concat mittels foldl definiert:

```
slow_concat :: [[a]] -> [a]
slow_concat = foldl (++) []
```

Zur Effizienz von `concat`, `slow_concat` (2)

...seien (vereinfachend) alle Listen `xsi` v. gleicher Länge `len`.

Dann hängen die (Kopier-) Kosten der Berechnung von

- ```
  concat [xs1,xs2,...,xsn]
   ->> foldr (++) [] [xs1,xs2,...,xsn]
   ->> xs1 ++ (xs2 ++ (... (xsn ++ [])) ...)
  ```

linear von der Anzahl n der Listen `xsi` ab: $n *$ `len` (jedes Konkatenieren erfolgt an eine Präfixliste der Länge `len`); die von

- ```
  slow_concat [xs1,xs2,...,xsn]
   ->> foldl (++) [] [xs1,xs2,...,xsn]
   ->> (... (([] ++ xs1) ++ xs2) ...) ++ xsn
  ```

hingegen quadratisch: $n * (n-1) *$ `len` (das Konkatenieren erfolgt an sukzessive länger werdende Präfixlisten: `0`, `len`, (`len`+`len`), (`len`+`len`+`len`),..., (`n-1`)∗`len`).

Zur Effizienz von concat, slow_concat (3)

... $n * (n-1) * \texttt{len}$ als Abschätzung der Summe:

$$
\begin{aligned}
& 0 \\
& + \texttt{len} \\
& + (\texttt{len} + \texttt{len}) \\
& + (\texttt{len} + \texttt{len} + \texttt{len}) \\
& \dots \\
& + \underbrace{(\texttt{len} + \texttt{len} + \dots + \texttt{len})}_{(n-1)-mal} \\
= & \sum_{i=1}^{n-1} i * \texttt{len} \\
= & \left(\sum_{i=1}^{n-1} i\right) * \texttt{len} \\
= & \frac{n*(n-1)}{2} * \texttt{len}
\end{aligned}
$$

Übungsaufgabe 10.5.2

Untersuchen und vergleichen Sie auch die Effizienz und Performanz der rekursiven Implementierung von reverse:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (l:ls) = (reverse ls) ++ [l]
```

...mit der Effizienz und Performanz der Implementierung aus dem Präludium:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
 where flip :: (a -> b -> c) -> (b -> a -> c)
       flip f x y = f y x
```

Zusammenfassung (1)

...die Bespiele zeigen, dass sich die Faltungsfunktionale `foldr` und `foldl` unterscheiden können hinsichtlich

- Anwendbarkeit (`foldr`, `foldl` mit Faltungsfunktionen `(flip (:))`, `(:)`)
- Effizienz (`foldr`, `foldl` mit Faltungsfunktion `(++)`)

...Eignung und Wahl von `foldr` und `foldl` sind deshalb problem- und kontextabhängig festzustellen und zu treffen!

Zusammenfassung (2)

...Programmierung mit Funktionalen macht

- das Wesen funktionaler Programmierung aus.

...unterstützt insbesondere

- Wiederverwendung von Programmcode.
- Kürzere und meist einfacher zu verstehende Programme.
- Einfachere Herleitung, einfacherer Beweis von Programmeigenschaften (Stichwort: Programmverifikation).
- ...

...vordefinierte Funktionale auf Listen leisten einen wesentlichen Beitrag hierzu.

Kapitel 10.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 10 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 6, Funktionen höherer Ordnung)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 5, Listen und Funktionen höherer Ordnung)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 5, Polymorphic and Higher-Order Functions; Kapitel 9, More about Higher-Order Functions)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 10 (2)

- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 7, Higher-order functions)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 5, Higher-order Functions)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 8, Funktionen höherer Ordnung)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 10 (3)

- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.5, Higher-order functional programming techniques)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 9.2, Higher-order functions: functions as arguments; Kapitel 10, Functions as values; Kapitel 19.5, Folding revisited)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 11, Higher-order functions; Kapitel 12, Developing higher-order programs; Kapitel 20.5, Folding revisited)

Kapitel 11

Polymorphie

Kapitel 11.1

Motivation

Polymorphie

Bedeutung lt. Duden:

- Vielgestaltigkeit, Verschiedengestaltigkeit

...mit verschiedenen fachspezifischen Bedeutungsausprägungen:

- Chemie: das Vorkommen mancher Mineralien in verschiedener Form, mit verschiedenen Eigenschaften, aber gleicher chemischer Zusammensetzung.
- Biologie: Vielgestaltigkeit der Blätter oder der Blüte einer Pflanze.
- Sprachwissenschaft: das Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. die verschiedenartigen Pluralbildungen in: die Tiere, die Felder, die Wiesen, die Pontons).
- Informatik, speziell Theorie der Programmiersprachen: das Thema dieses Kapitels.

Im programmiersprachlichen Kontext

...unterscheiden wir zwischen Polymorphie auf

- Datentypen (Kap. 11.2)
  - Algebraische Datentypen (`data`)
  - Neue Typen (`newtype`)
  - Typsynonyme (`type`)
- Funktionen (Kap. 11.3, Kap. 11.4)
  - Parametrische Polymorphie (oder echte Polymorphie)
    $\leadsto$ Sprachmittel: Typvariablen.
  - *Ad hoc* Polymorphie (oder unechte Polymorphie, Überladung)
    $\leadsto$ Haskell-spezifisches Sprachmittel: Typklassen.

Typvariablen in Haskell

...sind freigewählte Identifikatoren, die

- mit einem Kleinbuchstaben beginnen müssen (z.B.: `a`, `b`, `fp185A03`,...).

Beachte: Typnamen, Typ- und Datenwertkonstruktoren sind in Haskell ebenfalls freigewählte Identifikatoren, die im Unterschied zu Typvariablen

- mit einem Großbuchstaben beginnen müssen (z.B.: `A`, `B`, `String`, `Blatt`, `Wurzel`, `True`, `False`, `FP185A03`,...).

Kapitel 11.2

Polymorphie auf Datentypen

Polymorphe Datentypen

Definition 11.2.1 (Polymorpher Datentyp)

Ein algebraischer (Daten-) Typ, neuer Typ oder Typsynonym `T` heißt polymorph, wenn einer oder mehrere Grundtypen der Werte von `T` in Form einer oder mehrerer Typvariablen als Parameter angegeben werden.

Beispiele polymorpher Datentyp(deklaration)en:

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b c) c (Baum a b c)

newtype Paartripel a b c d = Pt ((a,b),(b,c),(c,d))

type Assoziationsliste a b = [(a,b)]
```

Polymorphe algebraische Datentypen (1)

Beispiele polymorpher algebraischer Datentypen:

```
data Liste a = Leer
             | Kopf a (Liste a)

data Baum a b c = Blatt a b
                | Wurzel (Baum a b c) c (Baum a b c)

data Graph a = Gph (a -> [a])

type Gewicht = Int
data GewichteterGraph a = GGph (a -> [(a,Gewicht)])

data Eq a => Liste' a = Leer'
                      | Kopf' a (Liste' a)

data (Eq a, Ord b, Ord c, Num c) => Baum' a b c
 = Blatt' a b
   | Wurzel' (Baum' a b c) c (Baum' a b c)
```

Polymorphe algebraische Datentypen (2)

Beispiele gültiger Listen- und Baumwerte:

```
Leer :: Liste a

Kopf 17 (Kopf 4 (Kopf (17+4) Leer)) :: Liste Int
Kopf 'a' (Kopf 'e' (Kopf 'i' (Kopf 'o' (Kopf 'u' Leer))))
                                                   :: Liste Char
Kopf True (Kopf (True&&False) (Kopf ((odd . fib) 42) Leer))
                                                   :: Liste Bool

Blatt "Fun Prog" 8 :: Baum [Char] Int c
Blatt True 3.14 :: Baum Bool Float c
Blatt 'a' 'z' :: Baum Char Char c

Wurzel (Blatt "Fun" 3) True (Blatt "Prog" 4)
                                        :: Baum [Char] Int Bool
Wurzel (Blatt "Fun" 3) (Kopf 42 Leer) (Blatt "Prog" 4)
                                 :: Baum [Char] Int (Liste Int)
Wurzel (Blatt "Fun" 3) Leer (Blatt "Prog" 4)
                                   :: Baum [Char] Int (Liste a)
```

Polymorphe algebraische Datentypen (3)

Beispiele gültiger Graph- und gewichteter Graphwerte:

```
data Knoten = K1 | K2 | K3 deriving Eq

g :: Knoten -> [Knoten]
g K1 = [K1,K2,K3]
g K2 = [K2,K3]
g K3 = []

g' :: Int -> [Int]
g' n = [n..2*n]

Gph g  :: Graph Knoten
Gph g' :: Graph Int

gg :: Knoten -> [(Knoten,Gewicht)]
gg K1 = [(K1,0),(K2,17),(K3,4)]
gg K2 = [(K2,0),(K3,42)]
gg K3 = []

gg' :: Int -> [(Int,Gewicht)]
gg' n = [(m,m+1) | m <- [n..2*n]]

GGph gg  :: GewichteterGraph Knoten
GGph gg' :: GewichteterGraph Int
```

Polymorphe neue Typen (1)

Beispiele polymorpher neuer Typen:

```
newtype Unverwechselbar_mit_Typ_a a = Uvbmta a

newtype Paar a = P (a,a)

newtype Paartripel a b c d = Pt ((a,b),(b,c),(c,d))

newtype Relation a b = R [(a,b)]

newtype Funktion a b = F (a->b)

newtype Ord a => Paar' a = P' (a,a)

newtype (Ord a, Ord b) => Relation' a b = R' [(a,b)]
```

Polymorphe neue Typen (2)

Beispiele gültiger neuer Typwerte:

```
Uvbmta 4711 :: Unverwechselbar_mit_Typ_a Int
Uvbmta Leer :: Unverwechselbar_mit_Typ_a (Liste a)
Uvbmta sqrt :: Unverwechselbar_mit_Typ_a (Float -> Float)

P (17,4)     :: Paar Int
P ([],[42]) :: Paar [Int]
P ([],[])   :: Paar [a]

Pt (("Fun",3),(4,odd(length "Prog"),(True,'T'))
        :: Unverwechselbar_mit_Typ_a [Char] Int Bool Char
Pt ((fac,'a'),('z',""),("Hallo, Welt!",id) )
        :: Unverwechselbar_mit_Typ_a (Integer -> Integer)
                                      Char [Char] (a -> a)

R [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] :: Relation Int Int
R [(Leer,42)] :: Relation (Liste a) Int
R [] :: Relation a b

F fac :: Funktion Integer Integer
F id  :: Funktion a a
F (\x->(\y->(x > length y))) :: Funktion Int ([a] -> Bool)
```

Polymorphe Typsynonyme (1)

Beispiele polymorpher Typsynonyme:

```
type Sequenz a = [a]

type Assoziationsliste a b = [(a,b)]

type MeinTyp a = Unverwechselbar_mit_Typ_a a

type MeinBaum a b c = Baum a b c

type MeinPaar a b = (Funktion a b,Relation a b)
```

Polymorphe Typsynonyme (2)

Beispiele gültiger Typsynonymwerte:

```
[] :: [a]                                   -- Sequenzwerte
[1,2,3,4,6,12] :: [Int]
[fac,fib,(+1)] :: [Integer -> Integer]

[] :: [(a,b)]                      -- Assoziationslistenwerte
[("Hallo,",6),(" ",1),("Welt!",5)] :: [([Char],Int)]
[(fac,"fac"),(fib,"fib"),((+1),"inc")]
                       :: [((Integer -> Integer),[Char])]

zahlwert      = Uvbmta 4711 :: MeinTyp Int  -- MeinTyp-Werte
listenwert    = Uvbmta Leer :: MeinTyp (Liste a)
funktionswert = Uvbmta sqrt :: MeinTyp (Float -> Float)
:t zahlwert      ->> Unverwechselbar_mit_Typ_a Int
:t listenwert    ->> Unverwechselbar_mit_Typ_a (Liste a)
:t funktionswert ->> Unverwechselbar_mit_Typ_a (Float -> Float)

baumwert = Blatt "Fun Prog" 8 :: MeinBaum [Char] Int Bool
:t baumwert ->> Baum [Char] Int c              -- MeinBaum-Wert

paarwert = (F (+1),R []) :: MeinPaar Int Int -- MeinPaar-Wert
:t paarwert ->> (Funktion Int Int,Relation Int Int)
```

Übungsaufgabe 11.2.2

Ergänze Beispiele gültiger Werte der Typen:

```
Liste' a
Baum' a b c

Paar' a
Relation' a b

MeinBaum' a b c
MeinPaar' a b
```

Kapitel 11.3

Parametrische Polymorphie auf Funktionen

Parametrisch polymorphe Funktionen

Definition 11.3.1 (Parametrisch polymorphe Fkt.)

Eine Funktion heißt parametrisch polymorph (oder echt polymorph), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen Werte beliebiger Typen als Argument zulassen.

Beispiele parametrisch polymorpher Funktionen:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
fst :: (a,b) -> a)
fst (x,_) = x
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
splitAt :: Int -> [a] -> [[a],[a]]
splitAt n xs = (take n xs,drop n xs)
```

Parametrische Polymorphie auf Funktionen

...tritt in funktionalen Sprachen beiläufig und ubiquitär auf:

- Die Funktionale `curry`, `uncurry`, `flip` und `id`.
- Die Funktionale `map`, `filter`, `foldl` und `foldr`.
- Die Funktionen `fst` und `snd`.
- Die Funktionen `length`, `head` und `tail`.
- ...

Die parametrisch polymorphen Funktionale

...curry, uncurry, flip und id auf beliebigen Typen:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry g (x,y) = g x y

flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x

id :: a -> a
id x = x
```

Anwendungsbeispiele:

```
id 3 ->> 3
id ["abc","def"] ->> ["abc","def"]
id fac 5 ->> (id fac) 5 ->> fac 5 ->> 120
```

Die parametrisch polymorphen Funktionale

...map, filter, foldl und foldr auf beliebigen Listentypen:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f l (foldr f e xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Die parametrisch polymorphen Funktionen

...fst und snd auf beliebigen Paartypen:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd(_,y) = y
```

Die parametrisch polymorphen Funktionen

...length, head und tail auf beliebigen Listentypen:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs

head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Die parametrisch polymorphen Funktionale

...zip und unzip ("Reißverschlussfunktionen") auf beliebigen Listentypen:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = []

zip [3,4,5] ['a','b','c','d'] ->> [(3,'a'),(4,'b'),(5,'c')]
zip ["abc","def","geh"] [(3,4),(5,4)]
                           ->> [("abc",(3,4)),("def",(5,4))]

unzip :: [(a,b)] -> ([a],[b])
unzip []          = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                    where
                    (xs,ys) = unzip ps

unzip [(3,'a'),(4,'b'),(5,'c')] ->> ([3,4,5],['a','b','c'])
unzip [("abc",(3,4)),("def",(5,4))]
                           ->> (["abc","def"],[(3,4),(5,4)])
```

Weitere vordefinierte parametrisch polymorphe

...Funktionale und Funktionen auf beliebigen Listentypen:

`(:)`	`::`	`a->[a]->[a]`	Listenkonstruktor (rechtsassoziativ)
`(!!)`	`::`	`[a]->Int->a`	Projektion auf i-te Komp., Infixop.
`length`	`::`	`[a]->Int`	Länge der Liste
`(++)`	`::`	`[a]->[a]->[a]`	Konkat. zweier Listen
`concat`	`::`	`[[a]]->[a]`	Konkat. mehrerer Listen
`head`	`::`	`[a]->a`	Listenkopf
`last`	`::`	`[a]->a`	Listenendelement
`tail`	`::`	`[a]->[a]`	Liste ohne Listenkopf
`init`	`::`	`[a]->[a]`	Liste ohne Endelement
`splitAt`	`::`	`Int->[a]->[[a],[a]]`	Aufspalten einer Liste an Position i
`reverse`	`::`	`[a]->[a]`	Umkehren einer Liste
...			

Parametrisch polymorphe Funktionen

...auf (selbstdefinierten) algebraischen Datentypen:

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b) c (Baum a b)

tiefe :: (Baum a b c) -> Int
tiefe (Blatt _ _) = 0
tiefe (Wurzel ltb _ rtb) = 1 + max (tiefe ltb) (tiefe rtb)

gen_assliste :: (Baum a b c) -> Assoziationsliste a b
gen_assliste (Blatt x y)          = [(x,y)]
gen_assliste (Wurzel ltb _ rtb) = (gen_assliste ltb)
                                   ++ (gen_assliste rtb)

plaetten :: (a -> b -> c) -> (Baum a b c) -> (Sequenz c)
plaetten f (Blatt x y)          = [f x y]
plaetten f (Wurzel ltb z rtb) = (plaetten f ltb)
                                 ++ [z]
                                 ++ (plaetten f rtb)
```

Einschränkung parametrischer Polymorphie

...durch Typkontexte bei der Funktionsdefinition:

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b c) c (Baum a b c)

wurzelsumme :: Num c => (Baum a b c) -> c
wurzelsumme (Blatt _ _) = 0
wurzelsumme (Wurzel ltb z rtb)
 = z + wurzelsumme ltb + wurzelsumme rtb
```

...bereits bei der Datentypdeklaration:

```
data Num c => Baum' a b c = Blatt' a b
                | Wurzel' (Baum' a b c) c (Baum' a b c)

wurzelsumme' :: Num c => (Baum' a b c) -> c
wurzelsumme' (Blatt' _ _) = 0
wurzelsumme' (Wurzel' ltb z rtb)
 = z + wurzelsumme' ltb + wurzelsumme' rtb
 -- Der Kontext bei wurzelsumme' ist trotz allem nötig.
```

Bemerkungen zur Typkontexteinschränkung

Die Einschränkung parametrischer Polymorphie auf numerische Typen als zulässige Instanzen der Typvariable `c` in der Signatur der Funktionen

- `wurzelsumme`, `wurzelsumme'`

ist nötig, weil sich

- beide Funktionen auf die (überladene) Funktion `(+)` abstützen, die ausschließlich für Werte numerischer Typen definiert ist, d.h. für Werte von Typen, die Element der Typklasse `Num` sind (siehe auch Kap. 11.4).

  (Bem.: Ohne Kontext wird für den Typ von `c` in `wurzelsumme'` der Typ `Integer` inferiert, nicht `c` ein Typ in `Num`.)

Zusammenfassung

Parametrische Polymorphie auf Funktionen ermöglicht Wiederverwendung von

- Funktionsnamen (Gute Namen sind knapp!)
- Funktionsrümpfen (d.h. Funktionsimplementierungen)

für beliebige Typen.

Nutzenveranschaulichung (1)

...anhand der Funktion `length` auf beliebigen Listentypen:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Dank parametrischer Polymorphie sind Aufrufe mit Listen über beliebigen Listenelementtypen unmittelbar möglich, z.B.:

```
length [2,4,23,2,53,4] ->> 6

length ["Enjoy","Functional","Programming"] ->> 3

length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] ->> 3

length [[2,4,23,2,5],[3,4],[],[56,7,6,12]] ->> 4

length [(Blatt 17 4), (Blatt 21 21),
        (Wurzel (Blatt 47 11) fac (Blatt 42 0))] ->> 3

length [fac,fib,fun91,(binom 45),(+1),(*2)] ->> 6
```

Nutzenveranschaulichung (2)

Ohne parametrische Polymorphie wäre für jeden Listentyp eine eigene Implementierung unter eigenem Namen nötig, die sich einzig in Namen und Typsignatur unterschieden:

```
length_Ganzzahlliste :: [Int] -> Int
length_Ganzzahlliste []     = 0
length_Ganzzahlliste (_:xs) = 1 + length_Ganzzahlliste t xs

length_Zeichenreihenliste :: [String] -> Int
length_Zeichenreihenliste [] = 0
length_Zeichenreihenliste (_:xs)
 = 1 + length_Zeichenreihenliste xs

...

length_Baumliste :: [Baum] -> Int
length_Baumliste []     = 0
length_Baumliste (_:xs) = 1 + length_Baumliste xs

length_Funktionsliste :: [(Integer -> Integer)] -> Int
length_Funktionsliste []     = 0
length_Funktionsliste (_:xs) = 1 + length_Funktionsliste xs
```

Nutzenveranschaulichung (3)

...mit folgenden argumenttypspezifischen Aufrufen:

```
length_Ganzzahlliste [2,4,23,2,53,4] ->> 6

length_Zeichenreihenliste
 ["Enjoy","Functional","Programming"] ->> 3

length_Gleitkommazahlpaarliste
  [(3.14,42.0),(56.1,51.3),(1.12,2.22)] ->> 3

length_Ganzzahllistenliste
  [[2,4,23,2,5],[3,4],[],[56,7,6,12]] ->> 4

length_Baumliste
 [(Blatt 17 4), (Blatt 21 21),
  (Wurzel (Blatt 47 11) fac (Blatt 42 0))] ->> 3

length_Funktionsliste
 [fac,fib,fun91,(binom 45),(+1),(*2)] ->> 6
```

Sprechweisen: Rechenvorschriften

...der Form

- `length :: [a] -> Int`

heißen (parametisch (oder echt) polymorph (definiert für alle Typen).

...der Form

- `length_Ganzzahlliste :: [Int] -> Int`
- `length_Zeichenreihenliste :: [String] -> Int`
- `length_Gleitkommazahlpaarliste ::`
  `[(Float,Float)] -> Int`
- `length_Ganzzahllistenliste :: [[Int]] -> Int`
- `length_Baumliste :: [Baum] -> Int`
- `length_Funktionsliste ::`
  `[(Integer -> Integer)] -> Int`

heißen monomorph (definiert für genau einen Typ).

Weitere Sprechweisen

...illustriert anhand der Typsignatur der Funktion `length`.

```
length :: [a] -> Int
```

Sprechweisen:

- `a` heißt Typvariable (Typvariablen werden mit Kleinbuchstaben gewöhnlich vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...).
- Typen wie

  ```
  [Int] -> Int
  [String] -> Int
  [(Float,Float)] -> Int
  [(Integer -> Integer)] -> Int
  ...
  ```

  heißen Instanzen des Typs `([a] -> Int)`, der selbst der allgemeinste Typ der Funktion `length` ist (siehe Kap. 14).

Hinweis: Das Hugs-Kommando :t

...liefert stets den (eindeutig bestimmten) allgemeinsten Typ eines wohlgeformten Haskell-Ausdrucks, wobei Typsynonyme zum Grundtyp hin aufgelöst werden (z.B. `[(a,b)]` und `[c]` statt `(Assoziationsliste a b)` und `(Sequenz c)`):

Beispiele:

```
Main>:t length
length :: [a] -> Int

Main>:t curry
curry :: ((a,b) -> c) -> (a -> b -> c)

Main>:t flip
flip :: (a -> b -> c) -> (b -> a -> c)

Main>:t gen_assoziationsliste
gen_assoziationsliste :: (Baum a b c) -> [(a,b)]

Main>:t plaetten
plaetten :: (a -> b -> c) -> (Baum a b c) -> [c]
```

Kapitel 11.4

Ad hoc Polymorphie auf Funktionen

Kapitel 11.4.1

Überladen von Funktionen, *ad hoc* Polymorphie

Ad hoc Polymorphie auf Funktionen

...tritt wie parametrische Polymorphie in funktionalen Sprachen beiläufig und ubiquitär auf:

- Die arithmetischen Funktionale `(+)`, `(*)`, `(-)`, etc.
- Die Booleschen Relatoren `(==)`, `(/=)`, `(>)`, `(>=)`, etc.
- ...

Die *ad hoc* polymorphen

...arithmetischen Funktionale und Funktionen in Haskell:

```
(+) :: Num a        => a -> a -> a
(*) :: Num a        => a -> a -> a
(/) :: Fractional a => a -> a -> a
div :: Integral a   => a -> a
```

...Booleschen Relatoren:

```
(==) :: Eq a  => a -> a -> Bool
(/=) :: Eq a  => a -> a -> Bool
(>)  :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
(<)  :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
```

...Operatoren und Relatoren:

```
min     :: Ord a => a -> a -> a
max     :: Ord a => a -> a -> a
compare :: Ord a => a -> a -> Ordering
```

Ad hoc Polymorphie

...ist eine schwächere, weniger generelle Form von Polymorphie mit folgenden Synonymen:

- Unechte Polymorphie.
- Überladen, Überladung (engl. Overloading).

Definition 11.4.1.1 (*Ad hoc* polym. Fkt. in Haskell)

Eine Funktion in Haskell heißt *ad hoc* polymorph (oder unecht polymorph oder überladen), wenn die Typen eines oder mehrerer ihrer Parameter angegeben durch Typvariablen durch Typkontexte eingeschränkt Werte aller durch den Typkontext zugelassenen Typen als Argument zulassen.

Beispiele *ad hoc* polymorpher Funktionen:

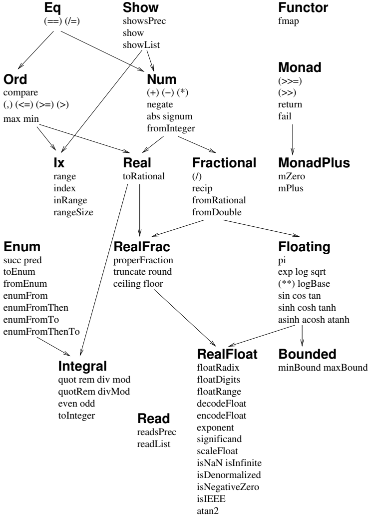
```
(+)  :: Num a => a -> a -> a
(==) :: Eq a  => a -> a -> Bool
(>)  :: Ord a => a -> a -> Bool
```

Wir unterscheiden

- vordefinierte (direkt) überladene Funktionen: Alle in einer vordefinierten Typklasse angegebenen Funktionen (z.B. `(==)`, `(/=)` aus `Eq`, `(<)`, `(>)` aus `Ord`, `(+)`, `(*)` aus `Num`, etc.)
- selbstdefinierte (direkt) überladene Funktionen: Alle in einer selbstdefinierten Typklasse angegebenen Funktionen (z.B. `auswertung`, `reihenausw`, `arithMittel` aus `Analysierbar`, `warnung`, `warnreihe` aus `Warnung` (vgl. Kap. 4.3))
- indirekt überladene Funktionen: Alle Funktionen, die sich auf eine überladene Funktion abstützen ohne selbst in einer Typklasse eingeführt zu sein, z.B.:

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

Vordefinierte Typklassen in Haskell: Erinnerung



Quelle: Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach. Addison-Wesley, 1999, Abb. 2.4.

Überladene Funktionen in vordef. Typklassen

...am Beispiel der Typklassen (Eq a) und (Num a):

```
class Eq a where
 (==), (/=) :: a -> a -> Bool      -- Signaturen über-
                                 -- ladener Funktionen
                                    -- in Typklasse Eq

 x /= y = not (x==y)                 -- Protoimplemen-
 x == y = not (x/=y)                      -- tierungen

class (Eq a,Show a) => Num a where
 (+), (-), (*) :: a -> a -> a      -- Signaturen über-
 negate :: a -> a                -- ladener Funktionen
 abs, signum :: a -> a             -- in Typklasse Num
 fromInteger :: Integer -> a
 x - y    = x + negate y             -- Protoimplemen-
 negate x = 0 - x                         -- tierungen
```

Allgemeines Muster

...einer Typklassendefinition (vereinfacht):

```
class Name' tv => Name tv where
 f_1 :: ...         -- Signaturen der Typklassen-
 f_2 :: ...             -- funktionen f_1,...,f_k
 ...                -- über tv und anderen Typen.
 f_k :: ...
 f_i = ...              -- Protoimplementierungen
 ...                       -- für 0 oder mehr der
 f_j = ...             -- Funktionen f_1,...,f_k.
```

Dabei:

- `Name'`: Name einer existierenden Typklasse als Kontext.
- `Name`: Freigewählter Name als Identifikator der Klasse.
- `tv`: Typvariable.

Ad hoc Polymorphie vs. Polymorphie

- Polymorphie: Ein polymorpher Typ wie `(a -> a)` steht informell für:

  $\forall$ `a` $\in$ `"Menge gültiger Typen". (a -> a)`

  Eine Funktion wie

  ```
  id :: a -> a
  ```

  ist für jeden gültigen Haskell-Typ eine Funktion vom Typ `(a -> a)`.

- *Ad hoc* Polymorphie: Ein *ad hoc* polymorpher Typ wie `(Num a => a -> a -> a)` steht informell für:

  $\forall$ `a` $\in$ `Num. (a -> a -> a)`

  Eine Funktion wie

  ```
  (+) :: Num a => a -> a -> a
  ```

  ist für jeden Typ, der Instanz der Typklasse `Num` ist, eine Funktion vom Typ `(a -> a -> a)`; für sonst keinen.

Überladene Funktionen in selbstdef. Typklassen

...am Beispiel der Typklassen (Info a) und (Groesse a):

```
class Info a where
 wert_beispiele  :: [a]          -- Signaturen überla-
 zu_zeichenreihe :: a -> String   -- dener Funktionen
                                 -- in Typklasse Info

 wert_beispiele    = []              -- Protoimplemen-
 zu_zeichenreihe _ = ""                   -- tierungen
           -- entspricht: zu_zeichenreihe = \_ -> ""

class Info a => Groesse a where
 groesse :: a -> Int                   -- Signatur über-
                                     -- ladener Funktion
                                -- in Typklasse Groesse

 groesse = (length . zu_zeichenreihe)   -- Protoimple-
                                         -- mentierung
```

Instanzbildungen für die Typen Char und Bool

...für die Typklassen (Info a) und (Groesse a):

```
instance Info Char where
 wert_beispiele     = ['a','A','z','Z','0','9']
 zu_zeichenreihe c = [c]

instance Groesse Char where
-- Die Protoimplementierung passte; nichts wäre zu
-- tun Aus Effizienzgründen geben wir dennoch an:
groesse _ = 1    -- entspricht: groesse = \_ -> 1

instance Info Bool where
 wert_beispiele          = [True,False]
 zu_zeichenreihe True  = "Wahr"
 zu_zeichenreihe False = "Falsch"

instance Groesse Bool where
 groesse True  = 4    -- length (zu_zeichenreihe True)
 groesse False = 6   -- length (zu_zeichenreihe False)
```

Instanzbildung für Typ Int

...für die Typklassen (Info a) und (Groesse a):

```
instance Info Int where
 wert_beispiele    = [-42..42]
 zu_zeichenreihe n = <Code, der einen Int-Wert
                      durch seine Ziffernfolge in
                      Binärdarstellungn als Zei-
                      chenreihe darstellt, z.B.
                      123 ↦ "1111011">

instance Groesse Int
-- Die Protoimplementierung passt; nichts zu tun.
-- (Das Schlüsselwort where kann hier entfallen.)
```

Instanzbildung für Typ [a]

...für die Typklassen (Info a) und (Groesse a):

```
instance Info a => Info [a] where
 wert_beispiele
   = [ [] ]
     ++ [ [x] | x <- wert_beispiele ]
     ++ [ [x,y] | x <- wert_beispiele,
                  y <- wert_beispiele ]
 zu_zeichenreihe = concat . (map zu_zeichenreihe)

instance Groesse a => Groesse [a] where
 groesse = ((foldr (+) 1) . (map groesse))
```

Beachte die überladene Verwendung der Funktionen:

- wert_beispiele, zu_zeichenreihe, groesse operieren auf Instanzen vom Typ [a].
- wert_beispiele, zu_zeichenreihe, groesse operieren auf Instanzen vom Typ a.

Instanzbildung für Typ (Baum a b c) (1)

...für die Typklassen (Info a) und (Groesse a):

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b) c (Baum a b)

instance (Info a,Info b,Info c) =>
                                Info (Baum a b c) where
 wert_beispiele
  = [Blatt (head wert_beispiele) (last wert_beispiele),
     Wurzel
      (Blatt (wert_beispiele!!0) (wert_beispiele!!1))
      (wert_beispiele!!2)
      (Blatt (wert_beispiele!!1) (wert_beispiele!!0))]
 zu_zeichenreihe baum
  = <Code, der einen Baum-Wert als eine (i.a. mehrzei-
    lige) Zeichenreihe darstellt, die die Baumstruktur
    durch Einrückungen und Zeilenumbrüche deutlich
    macht.>
```

Instanzbildung für Typ (Baum a b c) (2)

```
instance Info (Groesse a,Groesse b,Groesse c)
                        => Groesse (Baum a b c) where
 groesse = (length . zu_zeichenreihe)
 -- d.h. Übernahme der Protoimplementierung.
```

...oder:

```
instance Info (Groesse a,Groesse b,Groesse c)
                        => Groesse (Baum a b c) where
 groesse = tiefe
```

..oder:

```
instance Info (Groesse a,Groesse b,Groesse c)
                        => Groesse (Baum a b c) where
 groesse = ((2*) . length . gen_assliste)
```

...oder Größe als Summe von Blättern und Wurzeln, oder als Anzahl der {a,b,c}-Marken oder als...

Mittels des Hugs-Kommandos :t

...zur Bestimmung des allgemeinsten Typs eines gültigen Haskell-Ausdrucks erhalten wir:

```
Main> :t wert_beispiele
wert_beispiele :: Info a => [a]
Main> :t zu_zeichenreihe
zu_zeichenreihe :: Info a => a -> [Char]
Main> :t groesse
groesse :: Groesse a => a -> Int
```

Anwendungsbeispiele (1)

...der überladenen Funktionen der Typklassen Info und Groesse:

```
wert_beispiele :: Bool ->> [True,False]
wert_beispiele :: Char ->> ['a','A','z','Z','0','9']
[first (wert_beispiele :: Int)]
 ++ (last (wert_beispiele :: Int))   ->> [-42,42]
([first wert_beispiele] :: [Int])
 ++ ([last wert_beispiele] :: [Int]) ->> [-42,42]
zu_zeichenreihe True ->> "Wahr"
zu_zeichenreihe '5'  ->> "5"
zu_zeichenreihe 123  ->> "1111011"
```

Anwendungsbeispiele (2)

Abkürzungen: cc für concat, zz für zu_zeichenreihe.

```
zu_zeichenreihe [1,2,3]
 ->> zz [1,2,3]
 ->> (cc . (map zz)) [1,2,3]
 ->> cc (map zz [1,2,3])
 ->> cc [zz 1,zz 2,zz 3]
 ->> cc ["1","10","11"]
 ->> cc [['1'],['1','0'],['1','1']]
 ->> ['1','1','0','1','1']
 ->> "11011"
```

```
zu_zeichenreihe [[1,2,3],[4,5],[6]]
 ->> zz [[1,2,3],[4,5],[6]]
 ->> (cc . (map zz)) [[1,2,3],[4,5],[6]]
 ->> cc (map zz [[1,2,3],[4,5],[6]])
 ->> cc [zz [1,2,3],zz [4,5],zz [6]]
 ->> ...
 ->> cc ["11011","100101","110"]
 ->> ...
 ->> "11011100101110"
```

Anwendungsbeispiele (3)

...mit ergänzten Zwischenschritten.

```
zu_zeichenreihe [[1,2,3],[4,5],[6]]
 ->> zz [[1,2,3],[4,5],[6]]
 ->> (cc . (map zz)) [[1,2,3],[4,5],[6]]
 ->> cc (map zz [[1,2,3],[4,5],[6]])
 ->> cc [zz [1,2,3],zz [4,5],zz [6]]
 ->> cc [(cc . (map zz)) [1,2,3],(cc . (map zz)) [4,5],
         (cc . (map zz)) [6]]
 ->> cc [cc (map zz [1,2,3]),cc (map zz [4,5]),
         cc (map zz [6])]
 ->> cc [cc [zz 1,zz 2,zz 3],cc [zz 4, zz 5],cc [zz 6]]
 ->> cc [cc ["1","10","11"],cc ["100","101"],cc ["110"]]
 ->> cc [cc [['1'],['1','0'],['1','1']],
         cc [['1','0','0'],['1','0','1']],
         cc [['1','1','0']]]
 ->> cc [['1','1','0','1','1'],['1','0','0','1','0','1'],
         ['1','1','0']]
 ->> ['1','1','0','1','1','1','0','0','1','0','1','1','1','0']
 ->> "11011100101110"
```

Anwendungsbeispiele (4)

```
groesse False ->> 6
groesse 'z'   ->> 1
groesse 123
 ->> (length . zz) 123
 ->> length (zz 123)
 ->> length "1111011"
 ->> 7
groesse [[1,2,3],[4,5],[6]]
 ->> foldr (+) 1 (map (length . zz)) [[1,2,3],[4,5],[6]]
 ->> foldr (+) 1 [(length . zz) [1,2,3],(length . zz) [4,5],
                  (length . zz) [6]]
 ->> foldr (+) 1 [length (zz [1,2,3]),length (zz [4,5]),
                  length (zz [6])]
 ->> foldr (+) 1 [length "11011",length "100101",length "110"]
 ->> foldr (+) 1 [5,6,3]
 ->>             1 + foldr (+) 0 [5,6,3]
 ->>             1 + 14
 ->>             15
```

Übungsaufgabe 11.4.1.2

Ergänze fehlende Codestücke in den Beispielen:

- Instanzbildung (`Info Int`): Vervollständige die Implementierung der Funktion `zu_zeichenreihe`.
- Instanzbildung (`Info Baum a b c`): Vervollständige die Implementierung der Funktion `zu_zeichenreihe`.
- Instanzbildung (`Groesse Baum a b c`): Vervollständige die Implementierung d. Funktion `groesse` mit 'Größe' als
  - Summe von Blättern und Wurzeln,
  - Anzahl der {a,b,c}-Marken.

  Überlege mindestens eine weitere Variante, die als 'Größe' von Bäumen verstanden werden kann und nimm die zugehörige(n) Instanzbildung(en) für diese Variante(n) vor.

Teste die Implementierungen anhand geeigneter Beispiele, ob sie das gewünschte Verhalten zeigen; ebenso die in diesem Kapitel angegebenen Instanzbildungsimplementierungen.

Übungsaufgabe 11.4.1.3

Führe die Schritt- für Schrittauswertung der überladenen Funktionen `wert_beispiele`, `zu_zeichenreihe` und `groesse` der Typklassen `Info` und `Groesse` für weitere Werte aus, z.B. für Werte der Typen

- `String`, z.B. für die Werte `""` und `"abcd"`.
- `[Bool]`, z.B. für die Werte `[]` und `[True,False,True]`.
- `[[[Int]]]`, z.B. für die Werte `[]`, `[[]]`, `[[[]]]` und `[[[1,2,3],[4,5]],[[6],[7,8]]]`.
- `(Baum Int Int Int)`, z.B. für die Werte `(Blatt 17 4)` und `(Wurzel (Blatt 1 2) 3 (Blatt 4 5))`.
- ...

Übungsaufgabe 11.4.1.4

Mache weitere Typen zu Instanzen der Typklassen `Info` und `Groesse`, z.B. die Typen

- `Float`
- `Pegelstand` (s. Kap. 4)
- `Mensch` (s. Kap. 5)
- ...

Teste die Implementierungen auf gewünschtes Verhalten und führe auch hier für jeweils einige Datenbeispiele Schritt-für-Schritt-Auswertungen der überladenen Funktionen `wert_beispiele`, `zu_zeichenreihe` und `groesse` durch.

Zusammenfassung (1)

Typklassen sind

- Kollektionen von Typen, auf deren Werten die in der Typklasse angegebenen Funktionen definiert sind.

Durch Instanzbildungen der Typklasse für verschiedene Typen wird die Bedeutung

- dieser Funktionen überladen und typspezifisch.

Zweckmäßiger Weise (und auch üblicherweise) sind

- die typspezifischen Bedeutungen der überladenen Funktionen einander 'entsprechend', ihre Funktionalität einander 'vergleichbar', jeweils typspezifisch zugeschnitten.
- Instanzbildung mit 'vergleichbarer' Funktionalität kann nicht syntaktisch erzwungen werden; sie liegt in der Verantwortung des Programmierers und richtet einen Appell an die Programmierdisziplin.

Zusammenfassung (2)

Ad hoc Polymorphie (oder unechte Polymorphie oder Überladung) unterstützt Wiederverwendung des

- Funktionsnamens, nicht jedoch der Funktionsimplementierung (diese wird typspezifisch bei der Typklasseninstanzbildung ausprogrammiert, ggf. unter Ausnutzung der Protoimplementierungen der jeweiligen Typklasse):

```
(+)             :: Num a     => a -> a -> a
(>)             :: Ord a     => a -> a -> Bool
zu_zeichenreihe :: Info a    => a -> String
groesse         :: Groesse a => a -> Int
```

  d.h. es gilt das Prinzip:

  - ein Name, eine T-spezifische Implementierung pro Instanz T von a.

Zusammenfassung (3)

Parametrische (oder echte) Polymorphie unterstützt Wiederverwendung von

- Funktionsname und -implementierung:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

fst :: (a,b) -> a
fst (x,_) = x
```

d.h. es gilt das Prinzip:

  - ein Name, eine Implementierung.

Zusammenfassung (4)

Parametrische Polymorphie ist unter dem Aspekt Wiederverwendung echt stärker als *ad hoc* Polymorphie.

Dennoch: Bereits die von *ad hoc* Polymorphie unterstützte Wiederverwendung von Funktionsnamen ist äußerst nützlich.

Ohne *ad hoc* Polymorphie wären typspezifische Namen erforderlich nicht nur für

- eigendefinierte Funktionen ($\texttt{groesse}_{Int}$, $\texttt{groesse}_{Bool}$, $\texttt{groesse}_{Char}$, etc.)

sondern auch für bekannte Standardoperatoren wie

- Boolesche Relatoren ($=_{Bool}$, $>_{Float}$, $<_{String}$, etc.) und arithmetische Operatoren ($+_{Int}$, $-_{Float}$, $*_{Double}$, etc.)

Deren zwangweise Gebrauch wäre nicht nur ungewohnt und unschön, sondern in der täglichen Praxis auch äußerst lästig.

Anmerkung: Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg als Haskell und bieten andere Konzepte als Typklassen.

Kapitel 11.4.2

Vererben, erben, überschreiben

Vererben, erben, überschreiben

Typklassen können

- Spezifikationen mehr als einer Funktion bereitstellen.
- Protoimplementierungen (engl. default implementations) für (alle oder einige) dieser Funktionen bereitstellen.
- von anderen Typklassen erben.
- geerbte Implementierungen überschreiben.

In der Folge betrachten wir dies anhand einiger Beispiele in Haskell vordefinierter Typklassen.

Vererben, erben und überschreiben

...auf Typklassenebene:

```
class Eq a where
 (==), (/=) :: a -> a -> Bool
 x /= y = not (x==y)  -- Protoimplementierung f. (/=)
 x == y = not (x/=y)  -- Protoimplementierung f. (==)

class Eq a => Ord a where
 (<), (<=), (>), (>=) :: a -> a -> Bool
 max, min             :: a -> a -> a
 compare              :: a -> a -> Ordering
 x <= y = (x < y) || (x == y)    -- Protoimpl. f. (<=)
 x > y  = y < x                  -- Protoimpl. f. (<)
 ...
 compare x y       -- Protoimplementierung f. compare
   | x == y    = EQ
   | x <= y    = LT
   | otherwise = GT
```

Bemerkungen

- Die Typklasse `Ord` erweitert die Klasse `Eq`; jeder Typ, der zu einer Instanz der Typklasse `Ord` gemacht werden soll, muss bereits Instanz der Typklasse `Eq` sein.
- Jede Typinstanz von `Ord` erbt die Implementierungen ihrer Instanz aus `Eq`; umgekehrt vererbt jede Typinstanz aus `Eq` ihre Implementierungen an ihre Instanzen aus `Ord`.
- Für jede Typinstanz `T` der Typklasse `Ord` darf man sich darauf verlassen, dass es für `T`-Werte Implementierungen für alle Funktionen der Typklassen `Eq` und `Ord` gibt.
- Die Typklassen `Eq` und `Ord` stellen für einige Funktionen bereits Protoimplementierungen bereit.
- Für eine vollständige Instanzbildung reicht es deshalb, Implementierungen der Relatoren `(==)` und `(<)` anzugeben.
- Leisten die Protoimplementierungen nicht das Gewünschte, können sie durch instanzspezifische Implementierungen überschrieben werden.

Überschreiben automatisch generierter Impl.

...am Beispiel der Eq-Instanzbildung für den Typ (Paar a):

```
newtype Paar a = P (a,a)
instance (Eq a) => Eq (Paar a) where
 P (u,v) == P (x,y) = (u == x) && (v == y)
```

- Automatisch generiert: Die Eq-Instanz (Paar a) erhält für (/=) folgende sich aus den Protoimplementierungen der Klasse automatisch ergebende Implementierung:
  `P x /= P y = not (P x == P y)`
- Überschreiben: Die sich automatisch ergebende Implementierung von (/=) kann bei der Instanzbildung für (Paar a) überschrieben werden, z.B. durch folgende (geringfügig) effizientere Fassung:

```
instance (Eq a) => Eq (Paar a) where
 P (u,v) == P (x,y) = (u == x) && (v == y)
 P (u,v) /= P (x,y) = if u /= x then True else v /= y
```

Mehrfachvererben, -erben und -überschreiben

...auf Typklassenebene ist ebenfalls möglich; Haskells vordefinierte Typklasse Num ist ein Beispiel dafür:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a  -- Zwei Typkonver-
  fromInt       :: Int -> a      -- sionsfunktionen!

  x - y         = x + negate y      -- Protoimpl.
  fromInt       = ...
```

...jede Instanz der Typklasse Num muss auch Instanz der Typklassen Eq und Typklasse Show sein.

Übungsaufgabe 11.4.2.1

Vergleiche das Vererbungskonzept von Haskell mit dem Vererbungskonzept objektorientierter Sprachen, z.B. von Java.

Welche Gemeinsamkeiten, welche Unterschiede gibt es?

Kapitel 11.4.3

Automatische Typklasseninstanzbildung

Automatische Typklasseninstanzbildung (1)

...anhand von Beispielen:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                   deriving (Eq,Ord,Enum,Bounded,
                             Show,Read)

data Suchbaum = Leer | Knoten Suchbaum Integer Suchbaum
                 deriving (Eq,Ord,Show,Read)

newtype GanzeZahlen = GZ Int deriving (Eq,Ord,Bounded,
                                       Show,Read)

data (Ord a, Ord b, Ord c) => Baum a b c
 = Blatt a b
   | Wurzel (Baum a b c) c (Baum a b c)
                              deriving (Eq,Ord,Show)

newtype (Ord a, Ord b, Show a, Show b) =>
 Relation a b = R [(a,b)] deriving (Eq,Ord,Show)
```

Automatische Typklasseninstanzbildung (2)

Algebraische und neue Typen

- können mithilfe einer `deriving`-Klausel automatisch als Instanzen (einer festen Auswahl) vordefinierter Typklassen angelegt werden (keine Typsynonyme!).

Für die Funktionen der in der `deriving`-Klausel angeführten Typklassen wird dabei das

- "Offensichtliche" als Standardimplementierung generiert.

Intuitiv ersetzt die Angabe einer `deriving`-Klausel mit einer oder mehreren Typklassen

- die Angabe der entsprechenden `instance`-Direktiven.
- Elementare Typen (`Int`, `Float`, `Bool`, `Char`, etc.), Zeichenreihen (`String`) und Tupel und Listen solcher Typen sind vordefinierte Instanzen der infrage kommenden Typklassen (`Integer` z.B. nicht für die Typklasse `Bounded`).

Autom. vs. manuelle Typklasseninstanzbildung

Beispiel: Die Deklaration mit deriving-Klausel:

```
data (Eq a, Eq b, Eq c) => Baum a b c
 = Blatt a b
   | Wurzel (Baum a b c) c (Baum a b c) deriving Eq
```

ist gleichbedeutend zum Deklarationspaar:

```
data (Eq a, Eq b, Eq c) => Baum a b c
 = Blatt a b
   | Wurzel (Baum a b c) c (Baum a b c)

instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where
 (Blatt u v) == (Blatt x y) = (u == x) && (v == y)
 (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
   = (ltb == ltb') && (z == z') && (rtb == rtb')
 _ == _ = False
     -- Der Kontext ist nötig für die Instanzbildung.
```

Autom. vs. manuelle Typklasseninstanzbildung

...bzw. zum Deklarationspaar:

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b c) c (Baum a b c)

instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where
 (Blatt u v) == (Blatt x y) = (u == x) && (v == y)
 (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
   = (ltb == ltb') && (z == z') && (rtb == rtb')
 _ == _ = False
```

...mit "offensichtlicher" Gleichheit als Gleichheit in Struktur und Benennung.

Flexibilität durch manuelle Instanzbildung (1)

Manuelle Typklasseninstanzbildung erlaubt Gleichheit abweichend von "offensichtlicher" Gleichheit in (nahezu) jeder gewünschten Weise zu implementieren:

```
data Baum a b c = Blatt a b
                | Wurzel (Baum a b c) c (Baum a b c)
```

Z.B. als Gleichheit in Struktur und {a,c}-Benennungen, mit unbeachtet bleibenden (und möglicherweise unterschiedlichen) b-Benennungen:

```
instance (Eq a, Eq c) => Eq (Baum a b c) where
 (Blatt u _) == (Blatt x _) = (u == x)
 (Wurzel ltb z rtb) == (Wurzel ltb' z' rtb')
   = (ltb == ltb') && (z == z') && (rtb == rtb')
 _ == _ = False
```

Flexibilität durch manuelle Instanzbildung (2)

...oder als Gleichheit der Benennungen in mengenartigem Sinn:

```
instance (Ord a, Ord b, Ord c) => Eq (Baum a b c) where
 b == b' = (kollabiere b) == (kollabiere b')
```

wobei

```
kollabiere :: (Ord a, Ord b, Ord c) =>
                          (Baum a b c) -> ([a],[b],[c])
kollabiere = (entferne_duplikate . sortiere . aufsammeln)

aufsammeln :: (Baum a b c) -> ([a],[b],[c])
aufsammeln (Blatt x y) = ([x],[y],[])
aufsammeln (Wurzel ltb z rtb)
 = (aufsammeln ltb) +++ ([],[],[z]) +++ (aufsammeln rtb)

(+++) :: ([a],[b],[c]) -> ([a],[b],[c]) -> ([a],[b],[c])
(xs,ys,zs) +++ (xs',ys',zs') = (xs++xs',ys++ys',zs++zs')
```

Flexibilität durch manuelle Instanzbildung (2)

...und:

```
sortiere :: (Ord a,Ord b,Ord c)
                  => ([a],[b],[c]) -> ([a],[b],[c])
sortiere (xs,ys,zs)
  = (quickSort xs,quickSort ys,quickSort zs)

entferne_duplikate :: (Eq a,Eq b,Eq c)
                  => ([a],[b],[c]) -> ([a],[b],[c])
entferne_duplikate (xs,ys,zs)
  = <Code zum Entfernen von Duplikaten von Elementen>
```

Übungsaufgabe 11.4.3.1

Ergänze den Code für die Funktion `entferne_duplikate` und teste die verschiedenen `Eq`-Instantiierungsvarianten für den Datentyp (`Baum a b c`).

Automatische Typklasseninstanzbildung

...ist möglich (ausschließlich!) für folgende Menge vordefinierter Typklassen in Haskell:

- `Eq`
- `Ord`
- `Enum`
- `Bounded`
- `Show`
- `Read`

Für andere Typklassen, gleich ob vor- oder selbstdefiniert, sind zur Instanzbildung stets `instance`-Direktiven erforderlich; das gilt ebenso, falls von "offensichtlicher" abweichende Bedeutungen einer oder mehrerer Typklassenfunktionen gewünscht sind.

Kapitel 11.4.4

Grenzen des Überladens

Ist es möglich

...jeden Typ zu einer Instanz der Typklasse Eq zu machen?

De facto hieße das, den Typ des Gleichheitsrelators (==) von

```
(==) :: Eq a => a -> a -> Bool
```

über das Mittel des Überladens auf

```
(==) :: a -> a -> Bool
```

zu verallgemeinern; genauer, so nahe wie immer gewünscht daran anzunähern?

Im Sinne von

...Funktionen als erstrangigen Sprachelementen (engl. first class citizens) wäre ein Gleichheitstest auf Funktionen höchst wünschenswert, z.B.

```
(==) fac fib                         ->> False
(==) (\x -> x+x) (\x -> 2*x) ->> True
(==) (+2) (2+)                       ->> True
```

Anders als z.B. für die Parametervertauschung durch das Funktional

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

...ist Gleichheit eine typabhängige Eigenschaft, die eine typspezifische Implementierung erfordert.

In Haskell

...erforderte dies Eq-Instanzbildungen für funktionale Typen vorzunehmen, für die Abdeckung der Beispiele etwa für die Typen (Int -> Int) und (Int -> Int -> Int):

```
instance Eq (Int -> Int) where
(==) f g = ...

instance Eq (Int -> Int -> Int) where
(==) f g = ...
```

Preisfrage: Können wir die "Punkte" so ersetzen, dass wir eine valide Gleichheitsprüfung für alle Paare von Funktionen der Typen (Int -> Int) und (Int -> Int -> Int) erhalten?

Antwort: **Nein!**

Gleichheit von Funktionen: Unentscheidbar

Theorem 11.4.4.1 (Theoretische Informatik)

Gleichheit von Funktionen ist nicht entscheidbar, d.h. es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Beachte: Theorem 11.4.4.1 schließt nicht aus, dass für konkret vorgelegte Funktionen deren Gleichheit fallweise (algorithmisch) entschieden werden kann.

Zusammenfassung (1)

...anhand der Beobachtungen am Gleichheitsrelator:

- Funktionen bestimmter Funktionalität (auch scheinbar universeller Natur) lassen sich i.a. nicht für jeden Typ angeben, sondern nur für eine Teilmenge aller Typen.
- Die Funktionalität des Gleichheitsrelators ist ein konkretes Beispiel einer solchen Funktion.
- Auch wenn es verlockend wäre, eine
  - parametrisch polymorphe Implementierung des Gleichheitsrelators zu haben, in Haskell mit der Signatur

    ```
    (==) :: a -> a -> Bool
    ```

  ist eine Implementierung in dieser Allgemeinheit in keiner (!) Sprache möglich!

Zusammenfassung (2)

In Haskell

- sind die Typen, auf deren Werten der Gleichheitsrelator `(==)` definiert ist, genau die Elemente (oder Instanzen) der Typklasse `Eq`.

Bei der `Eq`-Instanzbildung für einen Typ `T` (gleich ob manuell oder automatisch) wird

- die exakte Bedeutung des Gleichheitsrelators für `T`-Werte durch die explizite Ausprogrammierung der Gleichheits- und Ungleichheitsrelatoren `(==)` und `(/=)` definiert und festgelegt.

Kapitel 11.5

Zusammenfassung

Polymorphie

...auf Funktionen und Datentypen unterstützt

- Wiederverwendung durch Parametrisierung

und damit die

- Ökonomie der Programmierung (flapsig: *Schreibfaulheit*).

durch Ausnutzung der Beobachtung, dass tragende Eigenschaften eines Datentyps wie von darauf arbeitenden Funktionen oft unabhängig von typspezifischen Details sind.

Insgesamt: Ein typisches Vorgehen in der Informatik:

- 'Gleiche' Teile werden 'ausgeklammert' und dadurch einer Wiederverwendung zugänglich gemacht.
- Im Fall von Polymorphie bedeutet das, dass ansonsten i.w. gleiche Codeteile nicht (länger) mehrfach geschrieben werden müssen.

Als Gratis-Nebeneffekt

...trägt Polymorphie bei zu höherer

- Transparenz und Lesbarkeit
  ...durch Betonung von Gemeinsamkeiten, nicht von Unterschieden.
- Verlässlichkeit und Wartbarkeit
  ...hinsichtlich Fehlersuche, Weiterentwicklung, etc.
- Programmiereffizienz
  ...hinsichtlich höherer Produktivität, früherer Markteintrittsmöglichkeit (engl. time-to-market).
- ...

Nichtzuletzt: Polymorphie

...ein aktuelles Forschungs- und Entwicklungsgebiet auch in anderen Paradigmen, speziell dem

- objektorientierter Programmierung.

Nutzen und Vorteile polymorpher Konzepte für Datentypen und Funktionen werden zunehmend auch für Datentypen und Methoden zu schätzen gewusst (Stich- und Schlagwort: Generic Java).

Zusammenfassend (1)

...über Teil IV “Funktionale Programmierung” der Vorlesung:

Die Stärken des funktionalen Programmierstils resultieren aus insgesamt wenigen Konzepten für

- Funktionen
- Datentypen

Tragend sind dabei die Konzepte von

- Funktionen als erstrangige Sprachelemente (engl. first class citizens)
  - Stichwort: Funktionen höherer Ordnung (Kap. 10)
- Polymorphie als durchgängiges Prinzip auf
  - Datentypen (Kap. 11.2)
  - Funktionen (Kap. 11.3, Kap. 11.4)

Zusammenfassend (2)

Kombination und nahtloses Zusammenspiel der tragenden (wenigen) Einzelkonzepte führen in Summe zur hohen

- Ausdruckskraft und Flexibilität des funktionalen Programmierstils.

  $\rightsquigarrow$ *Das Ganze ist mehr als die Summe seiner Teile!*

Zusammenfassend (3)

Speziell in Haskell tragen zu Ausdruckskraft und Flexibilität weitere paradigmen- und sprachspezifische Annehmlichkeiten zur automatischen Generierung bei, etwa von

- Listen: `[2,4..42]`, `[odd n | n <- [1..], n<1000]`.
- Selektorfunktionen: Verbundtyp-Syntax für algebraische Datentypen.
- Typklasseninstanzen: `deriving`-Klausel.

Siehe

- Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003.

für eine weiterführende und vertiefende Diskussion.

Kapitel 11.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 11 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 5, Polymorphic and Higher-Order Functions; Kapitel 9, More about Higher-Order Functions; Kapitel 12, Qualified Types; Kapitel 24, A Tour of Haskell's Standard Type Classes)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 11 (2)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 2, Believe the Type; Kapitel 7, Making our own Types and Type Classes)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 19, Formalismen 4: Parametrisierung und Polymorphie)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.8, Type classes and class methods)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 11 (3)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 12, Overloading and type classes; Kapitel 14.3, Polymorphic algebraic types; Kapitel 14.6, Algebraic types and type classes)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13, Overloading, type classes and type checking; Kapitel 14.3, Polymorphic algebraic types; Kapitel 14.6, Algebraic types and type classes)

Teil V

Fundierung funktionaler Programmierung

Kapitel 12

λ-Kalkül

Kapitel 12.1

Motivation

"...much of our attention is focused on functional programming, which is the most successful programming paradigm founded on a rigorous mathematical discipline. Its foundation, the lambda calculus, has an elegant computational theory and is arguably the smallest universal programming language. As such, the lambda calculus is also crucial to understand the properties of language paradigms other [than] functional programming..."

Exzerpt von der Startseite der "Programming Languages and Systems (PLS)" Forschungsgruppe an der University of New South Wales, Sydney, geleitet von Manuel Chakravarty und Gabriele Keller.
(http://www.cse.unsw.edu.au/~pls/PLS/PLS.html)

λ-Kalkül

...der λ-Kalkül ist

- ein formales Berechenbarkeitsmodell neben anderen wie
  - Turing-Maschinen
  - Markov-Algorithmen
  - Theorie rekursiver Funktionen
  - ...
- fundamental für die Berechenbarkeitstheorie.
- liefert formale Fundierung funktionaler Programmierung und Programmiersprachen.

Berechenbarkeitstheorie

...im Mittelpunkt stehende Fragen:

- Was heißt berechenbar?
- Was ist berechenbar?
- Wie aufwändig ist etwas zu berechnen?
- Gibt es Grenzen der Berechenbarkeit?
- ...

Intuitive Berechenbarkeit

…ein informeller Berechenbarkeitsbegriff.

"Etwas" ist intuitiv berechenbar

- wenn es eine irgendwie machbare effektive mechanische Methode gibt, die zu jedem gültigen Argument in endlich vielen Schritten den Funktionswert konstruiert und für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht.

Intuitiv berechenbar: Was ist damit gewonnen?

...für die Beantwortung der sehr konkreten Fragen der Berechenbarkeitstheorie zunächst einmal nichts, da die Bedeutung von

- intuitiv berechenbar vollkommen vage bleibt und nicht greifbar ist, ein Bauchgefühl:

  "...wenn es eine irgendwie machbare effektive mechanische Methode gibt..."

Formal berechenbar, formale Berechenbarkeit

Zentrale Aufgabe der Berechenbarkeitstheorie:

- Berechenbarkeitsbegriffe zu ersinnen und so zu konkretisieren, dass sie
  - formal gefasst,
  - einer präzisen Behandlung zugänglich gemacht und
  - bezüglich ihrer Ausdruckskraft und Stärke miteinander verglichen

  werden können.

Grundlegend und Ausgangspunkt dafür: Die Einführung

- formaler Berechnungsmodelle

die den Begriff "berechenbar" innerhalb des jeweiligen Modells präzise und rigoros zu definieren erlauben und damit

- handfeste Ausprägungen von Berechenbarkeit darstellen.

Formale Berechnungsmodelle (1)

...wichtige Beispiele:

- Turing-Maschinen
- Markov-Algorithmen
- Theorie rekursiver Funktionen
- ...
- λ-Kalkül

Formale Berechnungsmodelle (2)

...zeitlich eingeordnet:

- Allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936)
- Turing-Maschinen (Turing 1936)
- μ-rekursive Funktionen (Kleene 1936)
- Markov-Algorithmen (Markov 1951)
- Registermaschinen (Random Access Machines (RAMs)) (Shepherdson, Sturgis 1963)
- ...

Formale Berechnungsmodelle (3)

...(oberflächlich) charakterisiert.

Turing-Maschine(n)

- eine maschinenbasierte und maschinenorientierte Konkretisierung von Berechenbarkeit.

Markov-Algorithmen
Theorie rekursiver Funktionen
λ-Kalkül

- programmierbasierte und programmierorientierte Konkretisierungen von Berechenbarkeit.

Der λ-Kalkül (1)

...geht zurück auf Alonzo Church (1936).

Der λ-Kalkül

- ist eines neben anderen formalen Berechnungsmodellen.
- formalisiert einen Berechnungsbegriff über Paaren, Listen, Bäumen, auch potentiell unendlichen, über Funktionen höherer Ordnung, etc., und macht Berechnungen einfach ausdrückbar.
- zeichnet sich in diesem Sinne durch größere Praxisnähe als (einige) andere formale Berechnungsmodelle aus.

Der λ-Kalkül (2)

...ist über die Konkretisierung eines Begriffs von Berechenbarkeit hinaus wichtig und nützlich auch für andere Bereiche:

- Entwurf von Programmiersprachen und Programmiersprachkonzepten: Funktionale Programmiersprachen, Typsysteme, Polymorphie,...
- Semantik von Programmiersprachen: Denotationelle Semantik, Bereichstheorie (engl. domain theory),...
- Berechenbarkeitstheorie: Grenzen der Berechenbarkeit.

Die Church'sche These

Church'sche These

Eine Funktion ist genau dann intuitiv berechenbar, wenn sie λ-definierbar ist, d.h. im λ-Kalkül ausdrückbar.

Ein Beweis für diese These? Unmöglich! Aufgrund der Nichtfassbarkeit des Begriffs "intuitiv berechenbar" entzieht sich die Church'sche These jedem Beweisversuch.

Zur Church (-Turing) 'schen These siehe z.B.:

B. Jack Copeland. The Church-Turing Thesis. The Stanford Encyclopedia of Philosophy, 2002.
`http://plato.stanford.edu/entries/church-turing`

Gleichmächtigkeit von Berechnungsmodellen

...man hat jedoch folgendes beweisen können:

- Alle der eingangs genannten formalen Berechnungsmodelle sind gleich mächtig, d.h. was in einem Modell berechenbar ist, ist in jedem der anderen Modelle berechenbar und umgekehrt.

...dies kann als starker Hinweis (nicht als Beweis) darauf verstanden werden, dass

- jedes dieser formalen Berechnungsmodelle den Begriff intuitiver Berechenbarkeit wahrscheinlich "gut" (im Sinne von umfassend und vollständig) charakterisieren!

Allerdings

...dieser starke Hinweis schließt nicht aus, dass vielleicht schon morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit besser, umfassender und vollständiger charakterisierte.

Präzedenzfall: Primitiv rekursive Funktionen

- galten bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit.
- Jedoch: Echt schwächeres Berechnungsmodell.
- Beweis: Ackermann-Funktion: Berechenbar, aber nicht primitiv rekursiv darstellbar (Ackermann 1928).

(Zur Definition des Schemas primitiv rekursiver Funktionen siehe z.B.: Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 2.1.2.)

Die Ackermann-Funktion

...“berühmtberüchtigtes” Beispiel einer offensichtlich

- effektiv berechenbaren, insbesondere also intuitiv berechenbaren Funktion, jedoch nicht durch primitiv rekursive Funktionen.

Die Ackermann-Funktion in Haskell-Notation:

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
 | m == 0                = n+1
 | (m > 0) && (n == 0) = ack (m-1,1)
 | (m > 0) && (n /= 0) = ack (m-1,ack(m,n-1))
```

Intuitive Berechenbarkeit: Allgemein genug?

Orthogonal zur Frage einer

- angemessenen Formalisierung des Begriffs intuitiver Berechenbarkeit

...ist die Frage nach der

- Angemessenheit intuitiver Berechenbarkeit selbst.

Warum?

Die Auffassung intuitiver Berechenbarkeit als Existenzfrage

- "einer irgendwie machbaren effektiven mechanischen Methode, die zu jedem gültigen Argument in endlich vielen Schritten den Funktionswert konstruiert und für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht."

induziert eine

- funktionsorientierte Vorstellung von Algorithmus

die Berechnungsmodellen wie dem λ-Kalkül und anderen zugrundeliegt und weitergehend implizit die Problemtypen festlegt, die überhaupt als

- Berechenbarkeitsproblem

aufgefasst werden (können).

Beobachtung (1)

Aus Maschinensicht entspricht der funktionsorientierten Algorithmusauffassung eine

- stapelartige Verarbeitungs- und Berechnungssicht:

  Eingabe $\rightsquigarrow$ endl. Verarbeitung/Berechnung $\rightsquigarrow$ Ausgabe

die sich auch in der Arbeitsweise der Turing-Maschine findet.

Beobachtung (2)

Diese Sicht

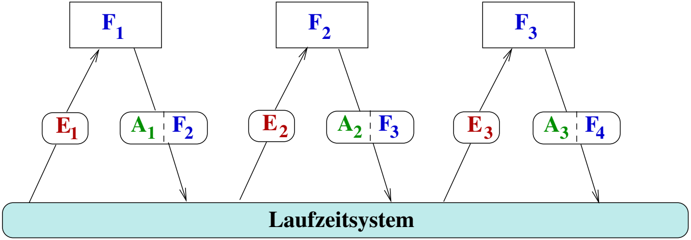
- findet sich in der Arbeitsweise früher automatischer Rechenanlagen (vulgo: Computer).
- entspricht auch der Auswertungsweise unserer bisherigen Haskell-Programme:

Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 245.

...Interaktion zwischen Anwender und Programm findet nach Bereitstellung der Eingabedaten in dieser Sicht nicht statt.

Beobachtung (3)

...Interaktion zwischen Anwender und Programm über die Bereitstellung von Eingabedaten hinaus ist für heutige konkrete Rechner jedoch kennzeichnend, auch für Haskell-Programme (siehe Kapitel 15, Ein- und Ausgabe):



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 253.

Naheliegende Fragen (1)

...sind Tätigkeiten, die

- Betriebssysteme
- Graphische Benutzerschnittstellen
- (Eingebettete) Steuerungssysteme
- Nebenläufige Systeme, Web-Services, das Internet
- ...

wahrnehmen oder Aufgaben wie

- Fahrzeuge autonom ihren Weg im realen Straßenverkehr zu vorgegebenen Zielen finden zu lassen
- ...

durch den funktionsorientierten Begriff intuitiver Berechenbarkeit gedeckt, d.h. vor- und darstellbar mit einmaliger Eingabedatenbereitstellung ohne weitere Interaktion?

Naheliegende Fragen (2)

...sind dies Probleme qualitativ anderer nichtfunktionaler Art?

Im Fall von Betriebssystemen:

- Ist die Berechnung, Verarbeitung endlich? Terminiert sie?
- Welche Funktion wird berechnet?

Im Fall autonomer Fahrzeuge:

- Wie sehen Ein- und Ausgabe aus?
- Welche Funktion wird berechnet?

Im Fall von Webservices, dem Internet:

- Können Systeme, in denen Komponenten hinzukommen und ebenso wieder verschwinden, in einem bestimmten Sinn als statisch angesehen werden?
- Welche Funktion wird berechnet?

...Interaktion scheint für Aufgaben dieser Art unverzichtbar.

Naheliegende Fragen (3)

...ändert Hinzunahme von Interaktion das Verständnis von Berechnung und Berechenbarkeit möglicherweise ähnlich grundlegend wie Ackermanns Funktion?

Angestoßen wurde diese Fragen und Untersuchungen hierzu besonders durch:

- Peter Wegner. Why Interaction is More Powerful Than Algorithms. Communications of the ACM 40(5):81-91, 1997.

Was heißt Berechnung, was berechenbar? (1)

Sind Antworten wie z.B. von

- Martin Davis. What is a Computation? Kapitel in L.A. Steeb (Hrsg.), Mathematics Today – Twelve Informal Essays. Springer-V., 1978.

...ausreichend oder bedürfen sie einer Anpassung vor dem Hintergrund von massiv parallelem, verteiltem Rechnen, Rechnen mit neuronalen Netzen, Quanten-Rechnern, interaktivem asynchronen Echtzeitrechnen, Nano-Rechnen, DNS-Rechnen,...?

- S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg). New Computational Paradigms: Changing Conceptions of What is Computable. Springer-V., 2008.

Was heißt Berechnung, was berechenbar? (2)

...im Sinn von Peter Wegner auf den Punkt gebracht: Gilt die Church/Turing-These

...im *schwachen* Sinn:

- Wann immer eine (mathematische) Funktion intuitiv berechenbar ist, d.h. wann immer es eine effektive mechanische Methode für ihre Berechnung gibt, dann kann sie von einer Turing-Maschine, im λ-Kalkül berechnet werden.

...oder im *starken* Sinn:

- Was immer eine "Berechnungsmaschine" ("Computer") berechnen kann, kann von einer Turing-Maschine, im λ-Kalkül berechnet werden, d.h. wann immer (über berechenbare Funktionen hinaus) eine Aufgabe als Berechnung ausgedrückt werden kann, kann sie von einer Turing-Maschine, im λ-Kalkül berechnet werden.

Offene Frage (1)

...die Church/Turing-These im starken Sinn salopp gefasst: Eine Aufgabe(nlösung) ist berechenbar, wenn sie von einer Turing-Maschine, im λ-Kalkül berechnet werden kann.

...eine vielfach "für" und "wider" untersuchte Frage:

- Michael Prasse, Peter Rittgen. Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability. The Computer Journal 41(6):357-362, 1998.
- Peter Wegner, Eugene Eberbach. New Models of Computation. The Computer Journal 47(1):4-9, 2004.
- Paul Cockshott, Greg Michaelson. Are There New Models of Computation? Reply to Wegner and Eberbach. The Computer Journal 50(2):232-247, 2007.
- Dina Q. Goldin, Peter Wegner. The Interactive Nature of Computing: Refuting the Strong Church-Rosser Thesis. Minds and Machines 18(1):17-38, 2008.

Offene Frage (2)

...hat Interaktion das Potential zu einer neuen Ackermannfunktion-ähnlichen Weltsichtänderung?

- Peter Wegner, Dina Q. Goldin. The Church-Turing Thesis: Breaking the Myth. In Proceedings of the 1st Conference on Computability in Europe – New Computational Paradigms (CiE 2005), Springer-V., LNCS 3526, 152-168, 2005.
- Martin Davis. The Church-Turing Thesis: Consensus and Opposition. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 125-132, 2006.

Offene Frage (3)

...Untersuchung und Diskurs gehen weiter; eigenes Verständnis und Einsicht in Voraussetzungen und Implikationen der "für"- und "wider"-Argumente sind gefordert.

Einen kompakten Einstieg in das Themenfeld zusammen mit Verweisen auf relevante Arbeiten bietet folgende Arbeit:

- B. Jack Copeland, Eli Dresner, Diane Proudfoot, Oron Shagrir. Viewpoint: Time to Reinspect the Foundations? Questioning if Computer Science is Outgrowing its Traditional Foundations. Communications of the ACM 59(11):34-36, 2016.

...weitere Literaturhinweise finden sich in Anhang A.

Zurück zum λ-Kalkül

Der λ-Kalkül ist ausgzeichnet durch:

- Einfachheit
  ...wenige syntaktische Konstrukte, einfache Semantik.
- Ausdruckskraft
  ...Turing-mächtig, alle "intuitiv berechenbaren" Funktionen im λ-Kalkül ausdrückbar.
- Bindeglied
  ...funktionaler Hochsprachen und maschinennaher Implementierungen.

Reiner λ-Kalkül, angewandte λ-Kalküle

Reiner λ-Kalkül

- Reduziert auf das “absolut Notwendige”, bedeutsam und besonders praktisch für Untersuchungen zur Fragen der Berechenbarkeit, Berechenbarkeitstheorie.

Angewandte λ-Kalküle

- Syntaktisch angereicherte Varianten des reinen λ-Kalküls, praxis- und programmiersprachennäher.

Extrem angereicherte angewandte λ-Kalküle

- Funktionale Programmiersprachen.

Kapitel 12.2

Syntax des λ-Kalküls

Reiner λ-Kalkül: Syntax von Ausdrücken (1)

Die Menge der wohlgeformten Ausdrücke des reinen λ-Kalküls über einer Menge N von Namen, kurz λ-Ausdrücke, ist mit E bezeichnet und wie folgt definiert:

- **Namen:** Jeder Name aus N ist in E.
  Bsp.: $a, b, c, \ldots, x, y, z, \ldots$

- **Abstraktion:** Ist $\mathtt{x}$ aus N und $\mathtt{e}$ aus E, so ist $(\lambda\mathtt{x}.\,\mathtt{e})$ in E.
  Sprechweise: (Funktions-) Abstraktion mit Parameter $\mathtt{x}$ und Rumpf $\mathtt{e}$.

  Bsp.: $(\overbrace{\lambda \mathtt{x}.\underbrace{(\mathtt{x\ x})}_{\text{Rumpf}}}^{\text{Abstraktion}})$, $(\overbrace{\lambda \mathtt{x}.\underbrace{(\overbrace{\lambda \mathtt{y}.\underbrace{(\overbrace{\lambda \mathtt{z}.\underbrace{(\mathtt{x\ (y\ z)})}})}}^{})}_{\text{Rümpfe}}}^{\text{Abstraktionen}})$,...

Reiner λ-Kalkül: Syntax von Ausdrücken (2)

- Applikation: Sind f und e aus E, so ist (f e) in E.

  Sprechweise: Anwendung von f auf e; f heißt auch Rator, e auch Rand.

  Bsp.: $(\underbrace{(\lambda \mathtt{x}.(\mathtt{x}\ \mathtt{x}))}_{\text{Rator}}\ \underbrace{\mathtt{y}}_{\text{Rand}}), \ldots$

Reiner λ-Kalkül: Syntax in BNF-Notation

...alternativ, die Ausdruckssyntax in Backus-Naur-Form (BNF):

$$
\begin{array}{lcll}
e & ::= & x & \text{(Namen)} \\
  & ::= & \lambda x.e & \text{(Abstraktion)} \\
  & ::= & e \; e & \text{(Applikation)} \\
  & ::= & (e) & \text{(Klammerung)}
\end{array}
$$

Vereinbarungen, Konventionen

Überflüssige Klammern können weggelassen werden. Es gilt:

- Rechtsassoziativität für λ-Sequenzen in Abstraktionen.

  Beispiele:
  – $\lambda x.\lambda y.\lambda z.(x\,(y\,z))$ steht kurz für $(\lambda x.(\lambda y.(\lambda z.(x\,(y\,z)))))$
  – $\lambda x.\, e$ steht kurz für $(\lambda x.\, e)$

- Linksassoziativität für Applikationssequenzen.

  Beispiele:
  – $e_1\ e_2\ e_3 \ldots e_n$ steht kurz für $(\ldots((e_1\ e_2)\ e_3)\ldots e_n)$
  – $e_1\ e_2$ steht kurz für $(e_1\ e_2)$

Der Rumpf einer λ-Abstraktion ist der längstmögliche dem Punkt folgende λ-Ausdruck.

Beispiel: $\lambda x.e\ f$ steht kurz für $\lambda x.(e\ f)$, nicht $(\lambda x.\ e)\ f$

Freie Variablen, gebundene Variablen (1)

...in λ-Ausdrücken. Sei a aus E:

Freie Variablen von a:

$$
\begin{array}{llll}
\mathrm{frei}(x) & = & \{x\} & \text{wenn } a \equiv x \text{ aus } N \\
\mathrm{frei}(\lambda x.e) & = & \mathrm{frei}(e) \setminus \{x\} & \text{wenn } a \equiv \lambda\, x.e \\
\mathrm{frei}(f\ e) & = & \mathrm{frei}(f) \cup \mathrm{frei}(e) & \text{wenn } a \equiv f\ e
\end{array}
$$

Gebundene Variablen von a:

$$
\begin{array}{llll}
\mathrm{gebunden}(x) & = & \emptyset & \text{wenn } a \equiv x \text{ aus } N \\
\mathrm{gebunden}(\lambda x.e) & = & \mathrm{gebunden}(e) \cup \{x\} & \text{wenn } a \equiv \lambda\, x.e \\
\mathrm{gebunden}(f\ e) & = & \mathrm{gebunden}(f) & \\
 & & \cup\ \mathrm{gebunden}(e) & \text{wenn } a \equiv f\ e
\end{array}
$$

Freie Variablen, gebundene Variablen (2)

Beispiel: Betrachte den λ-Ausdruck $((\lambda x.\ (x\ y))\ x)$.

Gesamtausdruck:

- x kommt in $((\lambda x.\ (x\ y))\ x)$ frei und gebunden vor.
- y kommt in $((\lambda x.\ (x\ y))\ x)$ frei vor, aber nicht gebunden.

Teilausdrücke:

- x kommt in $(\lambda x.\ (x\ y))$ gebunden vor, aber nicht frei.
- x kommt in $(x\ y)$ und (x) frei vor, aber nicht gebunden.
- y kommt in $(\lambda x.\ (x\ y))$, $(x\ y)$ und (y) frei vor, aber nicht gebunden.

Beachte: "Gebunden" ist nicht die Negation von "frei" (andernfalls gälte z.B. "x kommt gebunden in y vor").

Freie, gebundene Variablenvorkommen

...in λ-Ausdrücken:

- Definierende Vorkommen: Jedes Variablenvorkommen unmittelbar nach einem λ.
- Angewandte Vorkommen: Jedes nicht definierende Variablenvorkommen.
- Gebunden an: Relation zwischen Variablenvorkommen und definierenden Variablenvorkommen. Jedes Variablenvorkommen (gleich ob angewandt oder definierend) ist an höchstens ein definierendes Variablenvorkommen gebunden; definierende Vorkommen sind an ihr Vorkommen selbst gebunden.
- Freies Variablenvorkommen: Angewandtes Vorkommen, das an kein definierendes Vorkommen gebunden ist.
- Gebundenes Variablenvorkommen: Vorkommen (gleich ob angewandt oder definierend), das an ein definierendes Vorkommen gebunden ist.

Kapitel 12.3

Semantik des λ-Kalküls

Reiner λ-Kalkül: Semantik von Ausdrücken

...grundlegend für die Definition der Semantik von Ausdrücken des reinen λ-Kalküls, kurz λ-Ausdrücken, sind:

- Syntaktische Substitution
- Konversionsregeln / Reduktionsregeln

Syntaktische Substitution: Informell, intuitiv

...eine dreistellige Abbildung

$$\cdot[\cdot/\cdot] : E \to E \to V \to E$$

zur bindungsfehlerfreien Ersetzung frei vorkommender Variablen x durch einen Ausdruck e in einem Ausdruck e'.

Intuitiv: Angewendet auf zwei Ausdrücke e' und e und eine Variable x bezeichnet

$$e' [e/x]$$

denjenigen Ausdruck, der aus e' entsteht, indem jedes freie Vorkommen von x in e' durch e substituiert, ersetzt wird.

Beachte: Die vereinfachende intuitive Beschreibung nimmt keinen Bedacht auf mögliche Bindungsfehler. Freiheit von Bindungsfehlern stellt die formale Definition syntaktischer Substitution sicher.

Syntaktische Substitution: Formal

Die 3-stellige Abbildung syntaktische Substitution ist formal definiert durch:

$$\cdot[\cdot/\cdot] : E \to E \to V \to E$$

...angewendet auf Namensterme:

$$\mathsf{x}\,[\mathsf{e}/\mathsf{x}] = \mathsf{e}, \text{ wenn } \mathsf{x} \text{ aus } \mathsf{N}$$
$$\mathsf{y}\,[\mathsf{e}/\mathsf{x}] = \mathsf{y}, \text{ wenn } \mathsf{y} \text{ aus } \mathsf{N} \text{ mit } \mathsf{x} \neq \mathsf{y}$$

...angewendet auf applikative Terme:

$$(\mathsf{f}\ \mathsf{g})\,[\mathsf{e}/\mathsf{x}] = (\mathsf{f}\,[\mathsf{e}/\mathsf{x}])\ (\mathsf{g}\,[\mathsf{e}/\mathsf{x}])$$

...angewendet auf Abstraktionsterme:

$$(\,\lambda \mathsf{x}.\mathsf{f}\,)\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{x}.\mathsf{f}$$
$$(\,\lambda \mathsf{y}.\mathsf{f}\,)\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{y}.(\mathsf{f}\,[\mathsf{e}/\mathsf{x}]), \text{ wenn } \mathsf{x} \neq \mathsf{y} \text{ und } \mathsf{y} \notin \mathit{frei}(\mathsf{e})$$

$$(\,\lambda \mathsf{y}.\mathsf{f}\,)\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{z}.((\mathsf{f}\,[\mathsf{z}/\mathsf{y}])\,[\mathsf{e}/\mathsf{x}]), \text{ wenn } \mathsf{x} \neq \mathsf{y} \text{ und } \mathsf{y} \in \mathit{frei}(\mathsf{e}),$$
$$\text{wobei } \mathsf{z} \text{ frisch aus } \mathsf{N} \text{ mit } \mathsf{z} \notin \mathit{frei}(\mathsf{e}) \cup \mathit{frei}(\mathsf{f})$$

(Vermeidung von Bindungsfehlern!)

Beispiele

...zur Anwendung syntaktischer Substitution:

- $((x\ y)\ (y\ z))\ [(a\ b)/y] = ((x\ (a\ b))\ ((a\ b)\ z))$
- $\lambda x.\ (x\ y)\ [(a\ b)/y] = \lambda x.\ (x\ (a\ b))$
- $\lambda x.\ (x\ y)\ [(a\ b)/x] = \lambda x.\ (x\ y)$
- $\lambda x.\ (x\ y)\ [(x\ b)/y] \overset{naiv}{\leadsto} \lambda x.\ (x\ (x\ b))$: Bindungsfehler!
  ...naiv ohne Umbenennung angewendet ist x eingefangen!

Korrekt mit Umbenennung angewendet kein Bindungsfehler:

- $\lambda x.\ (x\ y)\ [(x\ b)/y] = \underbrace{\lambda z.\ ((x\ y)[z/x])}_{\text{Umbenennung von } x \text{ in } z}\ [(x\ b)/y]$

$$= \underbrace{\lambda z.\ (z\ y)}_{\text{Umbenannt}}\ [(x\ b)/y]$$

$$= \lambda z.\ (z\ (\underbrace{x}\ b))$$

Kein Bindungsfehler: x bleibt frei!

λ-Konversionsregeln, λ-Konversionen

...die λ-Konversionsregeln führen abgestützt auf syntaktische Substitution zur:

- α-Konversion (Umbenennung von Parametern)

  $$\lambda x.e \longleftrightarrow \lambda y.e\,[y/x],\ \text{wobei } y \notin \mathit{frei}(e)$$

- β-Konversion (Funktionsanwendung)

  $$(\,\lambda x.f\,)\ e \longleftrightarrow f\,[e/x]$$

- η-Konversion (Elimination redundanter Funktion)

  $$\lambda x.(e\ x) \longleftrightarrow e,\ \text{wobei } x \notin \mathit{frei}(e)$$

...und eine operationelle Semantik für λ-Ausdrücke.

Zur Anwendung der Konversionsregeln

α-Konversion

- zur konsistenten Umbenennung von Parametern von λ-Abstraktionen (zur Vermeidung von Bindungsfehlern!).

β-Konversion

- zur Anwendung einer λ-Abstraktion auf ein Argument.

η-Konversion

- zur Elimination redundanter λ-Abstraktionen.

Beachte: Naiv angewendet verursacht β-Konversion Bindungsfehler.

Bsp.: $(\lambda x.(\lambda y.x\ y))\ (y\ z) \longrightarrow (\lambda y.x\ y)[(y\ z)/x] \longrightarrow (\lambda y.(y\ z)\ y)$
(ohne α-Konversion ist y eingefangen: Bindungsfehler!)

...korrekt angewendet: Keine Bindungsf. dank α-Konversion!

Sprechweisen

...im Zusammenhang mit Konversionsregeln:

- Von links nach rechts angewendet: Reduktion.
- Von rechts nach rechts angewendet: Abstraktion.

Genauer:

- Von links nach rechts gerichtete Anwendungen der β- und η-Konversion heißen β-Reduktion und η-Reduktion.
- Von rechts nach links gerichtete Anwendungen der β-Konversion heißen β-Abstraktion.

Reduktionsfolgen, -strategien, Normalform

Eine Reduktionsfolge für einen λ-Ausdruck

- ist eine endliche oder nicht endliche Folge von β-, η-Reduktionen und α-Konversionen.
- heißt maximal, wenn höchstens noch α-Konversionen anwendbar sind.

Ein λ-Ausdruck ist in Normalform

- wenn er durch β-, η-Reduktion nicht weiter reduzierbar ist.

(Praktisch relevante) Reduktionsstrategien sind

- Normale (Reduktions-) Ordnung (leftmost-outermost)
- Applikative (Reduktions-) Ordnung (leftmost-innermost)

Beispiele (1)

...zu Reduktionsfolgen und -strategien.

Beispiel 1: Applikative Ordnung

$$(\underbrace{(\lambda z.\lambda y.(z\ y))}_{\text{Rator}}\ \underbrace{(\lambda x.x)}_{\text{Rand}})\ (\lambda s.(s\ s))$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad \underbrace{(\lambda y.((\lambda x.x)\ y))}_{\text{Rator}}\ \underbrace{(\lambda s.(s\ s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad \underbrace{(\lambda x.x)}_{\text{Rator}}\ \underbrace{(\lambda s.(s\ s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad \lambda s.(s\ s)$$

...fertig, Normalform erreicht: Keine β-, η-Reduktion mehr anwendbar.

Beispiele (2)

Beispiel 2: Applikative Ordnung

$$((\lambda x.\lambda y.x\ y)\ ((\underbrace{(\lambda x.\lambda y.x\ y)}_{\text{Rator}}\ \underbrace{a}_{\text{Rand}})\ b))\ c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\lambda x.\lambda y.x\ y)\ (\underbrace{(\lambda y.a\ y)}_{\text{Rator}}\ \underbrace{b}_{\text{Rand}}))\ c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\underbrace{(\lambda x.\lambda y.x\ y)}_{\text{Rator}}\ \underbrace{(a\ b)}_{\text{Rand}})\ c$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda y.(a\ b)\ y)}_{\text{Rator}}\ \underbrace{c}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow (a\ b)\ c$$

...fertig, Normalform erreicht: Keine β-, η-Reduktion mehr anwendbar.

Beispiele (3)

Beispiel 2': Normale Ordnung

$$(\underbrace{(\lambda x.\lambda y.x\ y)}_{\text{Rator}}\ \underbrace{(((\lambda x.\lambda y.x\ y)\ a)\ b)}_{\text{Rand}})\ c$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad \underbrace{(\lambda y.(((\lambda x.\lambda y.x\ y)\ a)\ b)\ y)}_{\text{Rator}}\ \underbrace{c}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad ((\underbrace{(\lambda x.\lambda y.x\ y)}_{\text{Rator}}\ \underbrace{a}_{\text{Rand}})\ b)\ c$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad (\underbrace{(\lambda y.a\ y)}_{\text{Rator}}\ \underbrace{b}_{\text{Rand}})\ c$$

$$(\beta\text{-Reduktion}) \quad \longrightarrow \quad (a\ b)\ c$$

...fertig, Normalform erreicht: Keine β-, η-Reduktion mehr anwendbar.

Existenz von Normalformen

...Normalformen existieren nicht notwendig; nicht jeder λ-Ausdruck ist

- besitzt oder ist in Normalform konvertierbar.

Beispiel:

- $\underbrace{\lambda x.(x\ x)}_{\text{Rator}}\ \underbrace{\lambda x.(x\ x)}_{\text{Rand}} \longrightarrow \lambda x.(x\ x)\ \lambda x.(x\ x) \longrightarrow \ldots$

  ...reproduziert sich endlos: Normalform existiert nicht!

Terminierung von Reduktionsfolgen

...Reduktionsfolgen terminieren nicht notwendig mit

- einem λ-Ausdruck in Normalform, selbst wenn eine Normalform existiert.

Beispiel:

- $\underbrace{(\lambda x.y)}_{\text{Rator}}\ \underbrace{(\lambda x.(x\ x)\ \lambda x.(x\ x))}_{\text{Rand}} \longrightarrow y$

  Normale Reduktionsordnung terminiert in einem Schritt: Normalform existiert!

- $(\lambda x.y)\ (\underbrace{\lambda x.(x\ x)}_{\text{Rator}}\ \underbrace{\lambda x.(x\ x)}_{\text{Rand}}) \longrightarrow (\lambda x.y)\ (\underbrace{\lambda x.(x\ x)}_{\text{Rator}}\ \underbrace{\lambda x.(x\ x)}_{\text{Rand}})$

  $\longrightarrow$ ...

  Applikative Reduktionsordnung terminiert nicht, obwohl Normalform existiert!

Church/Rosser-Theoreme

Seien e_1, e_2 zwei λ-Ausdrücke.

Theorem 12.3.1 (Konfluenz-, Diamant-, Rauteneig.)

Wenn e_1, e_2 ineinander konvertierbar sind, d.h. $e_1 \longleftrightarrow e_2$, dann gibt es einen gemeinsamen λ-Ausdruck e, zu dem e_1, e_2 reduziert werden können, d.h. $e_1 \longrightarrow^* e$ und $e_2 \longrightarrow^* e$.

Informell: Wenn eine Normalform existiert, dann ist sie (bis auf α-Konversion) eindeutig bestimmt!

Theorem 12.3.2 (Standardisierung)

Wenn e_1 zu e_2 mit einer endlichen Reduktionsfolge reduzierbar ist, d.h. $e_1 \longrightarrow^* e_2$, und e_2 in Normalform ist, dann führt auch die normale Reduktionsfolge von e_1 nach e_2.

Informell: Die normale Reduktionsordnung terminiert am häufigsten, so oft wie überhaupt möglich!

Folgerungen

...aus den Church/Rosser-Theoremen (Alonzo Church, John Barkley Rosser (1936)):

- Theorem 12.3.1 garantiert, dass die Normalform eines λ-Ausdrucks (bis auf α-Konversionen) eindeutig bestimmt ist, wenn sie existiert; λ-Ausdrücke in Normalform lassen sich (abgesehen von α-Konversionen) nicht mehr weiter reduzieren, vereinfachen.
- Theorem 12.3.2 garantiert, dass die normale Reduktionsordnung mit der Normalform terminiert, wenn es irgendeine Reduktionsfolge mit dieser Eigenschaft gibt, d.h. die normale Reduktionsordnung terminiert mindestens so häufig wie jede andere Reduktionsstrategie, mithin am häufigsten.

Semantik der Ausdrücke des reinen λ-Kalküls

...die Church/Rosser-Theoreme und ihre Garantien legen nahe, die Semantik (oder Bedeutung) der Ausdrücke des reinen λ-Kalküls in folgender Weise festzulegen:

Definition 12.3.3 (Semantik von λ-Ausdrücken)

Sei e ein λ-Ausdruck. Die Semantik von e ist

- seine (bis auf α-Konversionen) eindeutig bestimmte Normalform, wenn sie existiert; die Normalform ist zugleich der Wert von e.
- undefiniert, wenn die Normalform nicht existiert.

Determiniertheit, Turingmächtigkeit

Lemma 12.3.4 (Determiniertheit)

Wenn ein λ-Ausdruck in einen λ-Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktionsfolge des λ-Ausdrucks (bis auf α-Konversion) zu dieser Normalform, d.h. das Resultat jeder terminierenden Reduktionsfolge ist (bis auf α-Konversion) determiniert.

Theorem 12.3.5 (Turingmächtigkeit)

Eine Funktion ist im λ-Kalkül genau dann berechenbar, wenn sie Turing-berechenbar, Markov-berechenbar, etc., ist, d.h. im λ-Kalkül sind alle Funktionen berechenbar, die Turing-berechenbar, Markov-berechenbar, etc., sind und umgekehrt.

...nicht füreinander gemacht. Betrachte die rekursive Haskell-Rechenvorschrift `fac`:

Argumentbehaftet:

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Argumentfrei:

```
fac = λn. if n == 0 then 1 else n * fac (n - 1)
```

Im reinen λ-Kalkül stellt sich folgendes Problem:

- λ-Abstraktionen sind (im reinen λ-Kalkül wie in Haskell!) anonym und können deshalb nicht (rekursiv) aufgerufen werden.
- Rekursive Aufrufe wie im Rumpf von `fac` lassen sich deshalb nicht ohne weiteres im reinen λ-Kalkül ausdrücken.

Kunstgriff: Kombinatoren, Y-Kombinator

Kombinatoren

- sind spezielle λ-Terme, λ-Terme ohne freie Variablen.

Der Y-Kombinator (mit Selbstanwendung!):

- `Y` $=$ λ`f.(`λ`x.(f (x x))` λ`x.(f (x x)))`

Schlüsseleigenschaft des Y-Kombinators: Selbstreproduktion!

- Für e λ-Ausdruck ist (Y e) zu (e (Y e)) konvertierbar:

$$
\begin{array}{lcl}
\mathtt{Y\ e} & \longleftrightarrow & \underbrace{\mathtt{(\lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x))))}}_{=\mathsf{Y}}\ \mathtt{e} \\
 & \longrightarrow & \underbrace{\mathtt{\lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x))}}_{=\mathsf{Y\ e}} \\
 & \longrightarrow & \underbrace{\mathtt{e\ (\lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x)))}}_{=\mathsf{e\ (Y\ e)}} \\
 & \longleftrightarrow & \mathtt{e\ (Y\ e)}
\end{array}
$$

...Selbstreproduktion plus Kopie des Arguments e!

Der Y-Kombinator

...erlaubt Rekursion auf

- Kopieren zurückzuführen und zu realisieren.

Idee: Überführe eine rekursive Darstellung von f in eine nicht-rekursive Darstellung, die den Y-Kombinator verwendet:

$$
\begin{array}{llr}
& f = \cdots f \cdots & \text{(Rekursive Darstellung von f)} \\
\leadsto & f = \lambda f.\, (\cdots f \cdots)\; f & \text{(λ-Abstraktion)} \\
\leadsto & f = \underbrace{Y\; \lambda f.\, (\cdots f \cdots)}_{\text{“}Y\ e\text{”}} & \text{(Nichtrekursive Darstellung von f)}
\end{array}
$$

Übung: Vergleiche den Effekt des Y-Kombinators mit der Kopierregelsemantik prozeduraler Programmiersprachen.

Übungsaufgabe 12.3.6

Anwendung des Y-Kombinators

Betrachte die rekursionsfreie Darstellung von `fac` mit Y-Kombinator:

$$\mathsf{fac} = \mathsf{Y}\ \lambda \mathsf{f}.\,(\lambda \mathsf{n}.\,\mathsf{if}\ \mathsf{n} == 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{n} * \mathsf{f}\ (\mathsf{n}\ \text{-}\ 1))$$

Zeige durch nachrechnen, dass sich der Term `(fac 1)` auf den in Normalform vorliegenden Term 1 reduzieren lässt:

- $\mathsf{fac}\ 1 \longrightarrow\ \ldots\ \longrightarrow 1$

Überprüfe dabei, dass durch den

- Y-Kombinator Rekursion auf wiederholtes Kopieren zurückgeführt wird.

Angewandte λ-Kalküle

...sind syntaktisch angereicherte Varianten des reinen λ-Kalküls.

In Ausdrücken angewandter λ-Kalküle können

- Konstanten, Funktionsnamen, "übliche" Operatoren ähnlich wie Namen auftreten und an die Seite von λ-Abstraktionen treten:

  42, 3.14, true, false, $+$, $*$, $-$, fac, binom, $\ldots$
- neue Ausdrücke als Abkürzungen eingeführt und verwendet werden:

  cond e e_1 e_2 , if e then e_1 else e_2, $\ldots$
- Typen auftreten, Ausdrücke getypt sein:

  $42 : \mathbb{N}$, $3.14 : \mathbb{R}$, true : $\mathbb{B}\mathrm{ool}$, $\ldots$
- ...

Wohlgeformte Ausdrücke

...angewandter λ-Kalküle können dann auch

- Applikative Terme wie
  $2+3$, fac 3, fib $(2+3)$, binom x y, ((binom x) y), ...
- Abstraktionsterme wie
  $\lambda x.\,(x + x)$, $\lambda x.\lambda y.\lambda z.\,(x * (y - z))$,
  $(\lambda x.$ if odd x then $x * 2$ else x div $2)$, ...

sein, für deren Auswertung zusätzliche Reduktionsregeln eingeführt werden:

- δ-Reduktionen

...zur Auswertung, Reduktion arithmetischer Ausdrücke, bedingter Ausdrücke, Listenoperationen, etc.

Beispiel

...δ-Reduktionsfolge: Unecht "applikative" Ordnung
(unecht, da ohne Verzahnung von β-, η- und δ-Reduktionen)

$(\lambda x. \lambda. x * y)\ ((\lambda x. \lambda y. x + y)\ 9\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y)\ ((\lambda y. 9 + y)\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y)\ (9 + 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda y. (9 + 5) * y)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (9 + 5) * 3$

...keine β-, η-Reduktion mehr anwendbar; weiter mit δ-Reduktionen:

$\ldots\ (9 + 5) * 3$

$(\delta\text{-Reduktion, li}) \longrightarrow 14 * 3$

$(\delta\text{-Reduktion, li}) \longrightarrow 42$

Anm.: Ratoren in rot, Randen in gold; li für leftmost-innermost.

Beispiel′

...δ-Reduktionsfolge: Applikative Ordnung

$(\lambda x.\lambda.\, x{*}y)\ ((\lambda x.\lambda y.\, x{+}y)\ 9\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x.\lambda y.\, x{*}y)\ ((\lambda y.\, 9 + y)\ 5)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda x.\lambda y.\, x{*}y)\ (9 + 5)\ 3$

$(\delta\text{-Reduktion, li}) \longrightarrow (\lambda x.\lambda y.\, x{*}y)\ 14\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow (\lambda y.\, 14 * y)\ 3$

$(\beta\text{-Reduktion, li}) \longrightarrow 14 * 3$

$(\delta\text{-Reduktion, li}) \longrightarrow 42$

Anm.: Ratoren in rot, Randen in gold; li für leftmost-innermost.

Übungsaufgabe 12.3.7

δ-Reduktionsfolgen

Gegeben sei der λ-Ausdruck e:

$(\lambda x.\lambda.\, x{*}y)\ ((\lambda x.\lambda y.\, x{+}y)\ 9\ 5)\ 3$

Ergänze die applikative δ-Reduktionsfolge für diesen Ausdruck aus dem vorhergehenden Beispiel um

- die normale δ-Reduktionsfolge.
- zwei weitere zur Normalform führende δ-Reduktionsfolgen verschieden von der applikativen und normalen Folge.
- Gibt es darüberhinaus weitere verschiedene zur Normalform führende Reduktionsfolgen für e?

Typisierte λ-Kalküle

...ordnen jedem wohlgeformten Ausdruck einen Typ zu, zum Beispiel:

$$\begin{aligned} 3 &: \mathsf{Int} \\ (*) &: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ (\lambda x.2 * x) &: \mathsf{Int} \to \mathsf{Int} \\ (\lambda x.2 * x)\ 3 &: \mathsf{Int} \end{aligned}$$

Dabei treten zwei Schwierigkeiten auf:

1) Ausdrücke mit Selbstanwendung (wie z.B. der Y-Kombinator) können
   - nicht endlich getypt werden, d.h. ihr Typ kann nicht durch einen gewöhnlichen endlichen Typausdruck beschrieben werden.

2) Ausdrücke wie der Y-Kombinator können unmittelbar
   - nicht zur Modellierung von Rekursion verwandt werden.

Typisierung, Selbstanwendung, Rekursion

...Selbstanwendung im Y-Kombinator:

- $\mathsf{Y} = \lambda \mathsf{f}.\,(\lambda \mathsf{x}.\,(\mathsf{f}\ (\mathsf{x}\ \mathsf{x}))\ \lambda \mathsf{x}.\,(\mathsf{f}\ (\mathsf{x}\ \mathsf{x})))$

verhindert

- endliche Typisierung von Y in gewöhnlicher Weise.
- die Modellierung von Rekursion durch kopieren mithilfe von Y.

Überwindung

...der Typisierungsschwierigkeit:

- Rigoros: Übergang zu mächtigeren Typsprachen (Bereichstheorie, reflexive Bereiche (engl. domain theory, reflexive domains)).

...der Rekursionsschwierigkeit:

- Pragmatisch: Explizite Hinzunahme der Reduktionsregel

  $$Y\ e \longrightarrow e\ (Y\ e)$$

  zum Kalkül.

Rechtfertigung, Fundierung

...des Umgangs mit angereicherten angewandten λ-Kalkülen und des pragmatischen Vorgehens der Reduktionsregelhinzunahme für Rekursion:

Resultate aus der theoretischen Informatik, insbesondere

- Alonzo Church. The Calculi of Lambda-Conversion. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941

  ...u.a. zur Modellierung ganzer Zahlen, Wahrheitswerten, etc. durch Ausdrücke des reinen λ-Kalküls.

Bemerkungen

- Der übergang zu angewandten λ-Kalkülen ist aus praktischer Hinsicht sinnvoll und einsichtig; für theoretische Untersuchungen zur Berechenbarkeit (Berechenbarkeitstheorie) sind sie kaum relevant.
- Die Regelhinzunahme zur Rekursionsmodellierung ist aus Effizienzgründen auch pragmatisch zweckmäßig.

Zusammenfassung

- Haskell beruht auf den Grundlagen typisierter λ-Kalküle.
- Übersetzer, Interpretierer prüfen, ob die Typisierung von Haskell-Programmen konsistent, wohlgetypt ist.
- Programmierer können Typdeklarationen angeben (aussagekräftigere Fehlermeldungen, Sicherheit), müssen aber nicht (bequem, doch u.U. mit unerwarteten Folgen, etwa bei "zufällig" korrekter, aber "ungemeinter" Typisierung: "gemeinte" Typisierung wäre bei Angabe bei der Typprüfung als inkonsistent aufgefallen).
- Typinformation (gleich ob angegeben oder nicht) wird vom Übersetzer, Interpretierer berechnet, inferiert.
- Rekursion kann unmittelbar ausgedrückt werden (Y-Kombinator nicht erforderlich).

Zum Schluss

...folgende Anekdote zur Entwicklung der λ-Notation anhand der durch eine abstrakte λ-Abstraktion definierten Funktion:

```
fac :: Integer -> Integer
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

In Haskell also abweichend vom λ-Kalkül Verwendung von

- "\" und "->" anstelle von "λ" und "."

...der Weg dorthin war kurvenreich:

	$(\widehat{n.\, n+1})$	(Churchs handschriftliche Schreibweise)
$\leadsto$	$(\wedge n.\, n+1)$	(Churchs notat. Zugeständnis an Schriftsetzer)
$\leadsto$	$(\lambda n.\, n+1)$	(Pragmatische Umsetzung durch Schriftsetzer)
$\leadsto$	`(\n -> n+1)`	(ASCII-Umsetzung in Haskell)

(siehe: Peter Pepper. Funktionale Programmierung in Opal, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003, S. 22.)

Kapitel 12.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (1)

- Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 233-246, 1995.

- Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North-Holland, 1984. (Kapitel 1, Introduction; Kapitel 2, Conversion; Kapitel 3, Reduction; Kapitel 6, Classical Lambda Calculus; Kapitel 11, Fundamental Theorems)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (2)

- Hendrik P. Barendregt, Erik Barendsen. *Introduction to the Lambda Calculus*. Revised Edn., Technical Report, University of Nijmegen, March 2000. `ftp://ftp.cs.kun.nl/pub/CompMath.Found/lambda.pdf`

- Henrik P. Barendregt, Wil Dekkers, Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2012.

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 19, Berechenbarkeit und Lambda-Kalkül)

- Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (3)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 5, Lambda Calculus)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 4, Der Lambda-Kalkül)
- Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 6, Mathematical foundations: the lambda calculus)
- Robert M. French. *Moving Beyond the Turing Test*. Communications of the ACM 55(12):74-77, 2012.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (4)

- Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 1, Introduction; Kapitel 2, Notation and the Basic Theory; Kapitel 3, Reduction; Kapitel 10, Further Reading)
- A. Jung. *Berechnungsmodelle*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 73-88, 2006. (Kapitel 2.1, Speicher-orientierte Modelle: Turing-Maschinen, Registermaschinen; Kapitel 2.2, Funktionale Modelle: Algebraische Kombinationen, Primitive Rekursion, μ-Rekursion, λ-Kalkül)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (5)

- Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009. (Kapitel 2.1, Berechenbare Funktionen; Kapitel 2.2, Der λ-Kalkül)
- John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Electronic Notes in Theoretical Computer Science 1:370-392, 1995.
- John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. Journal of Functional Programming 8(3):275-317, 1998.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (6)

- John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Theoretical Computer Science 228(1-2):175-210, 1999.

- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 2, Lambda calculus; Kapitel 4.1, Repetition, iteration and recursion; Kapitel 4.3, Passing a function to itself; Kapitel 4.6, Recursion notation; Kapitel 8, Evaluation)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (7)

- William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 9, Formalismen 1: Zur Semantik von Funktionen)
- Gordon Plotkin. *Call-by-name, Call-by-value, and the λ-Calculus*. Theoretical Computer Science 1:125-159, 1975.
- Uwe Schöning, Wolfgang Thomas. *Turings Arbeiten über Berechenbarkeit – eine Einführung und Lesehilfe*. Informatik Spektrum 35(4):253-260, 2012. (Abschnitt Äquivalenz zwischen Turingmaschinen und Lambda-Kalkül)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (8)

- Allen B. Tucker (Editor-in-Chief). *Computer Science. Handbook*. Chapman & Hall/CRC, 2004.
  (Kapitel 92.3, The Lambda Calculus: Foundation of All Functional Languages)
- Ingo Wegener. *Grenzen der Berechenbarkeit*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 111-118, 2006. (Kapitel 4.1, Rechnermodelle und die Churchsche These)

Kapitel 13

Auswertungsordnungen

Applikative und normale Auswertungsordnung

...im Kontext von Haskell, speziell

- "Alias"-Charakterisierungen.
- Argumentauswertung bei Funktionsaufrufen.
- Operationalisierungen in Form
  - linksapplikativer (engl. eager)
  - linksnormaler
  - verzögerter (engl. lazy)

  Auswertung.

Kapitel 13.1

Motivation

"Aliase" applikativer und normaler Auswertung

Applikative Auswertungsordnung (engl. applicative order eval.)

- Verwandte Bezeichnungen: Wertparameter-, innerste oder strikte Auswertung, sofortige (oder unverzügliche) (Argument-) Auswertung (engl. call-by-value, innermost or strict evaluation).
- Operationalisierung: Linksapplikative, linkestinnerste oder sofortige Auswertung (engl. leftmost-innermost or eager evaluation).

Normale Auswertungsordnung (engl. normal order evaluation)

- Verwandte Bezeichnungen: Namensparameter-, äußerste Auswertung, aufgeschobene (oder verzögerte) (Argument-) Auswertung (engl. call-by-name, outermost evaluation).
- Operationalisierung: Linksnormale, linkestäußerste Auswertung (engl. leftmost-outermost evaluation).
- Effiziente Operationalisierungsumsetzung: Verzögerte Auswertung (engl. lazy evaluation).
  - Verwandt: Bedarfspar.-Ausw. (engl. call-by-need eval.).

Argumentauswertung in Funktionsaufrufen (1)

Applikativ: Unverzügliche Argumentauswertung.

- Ein applikativer Ausdruck `(f exp`$_1$` ... exp`$_n$`)` wird ausgewertet, indem
  - zunächst die Argumentausdrücke $\texttt{exp}_1,\ldots,\texttt{exp}_n$ vollständig ausgewertet werden, anschließend ihre Werte $\texttt{v}_1,\ldots,\texttt{v}_n$ im Rumpf von `f` für die Parameter von `f` eingesetzt werden und schließlich der so entstandene expandierte Ausdruck ausgewertet wird.

Die Wirkung unverzüglicher Argumentauswertung motiviert folgende Bezeichnungsvarianten:

- Wertparameter-, innerste, strikte Auswertung (engl. call-by-value evaluation, innermost evaluation, strict evaluation).

Argumentauswertung in Funktionsaufrufen (2)

Normal: Aufgeschobene Argumentauswertung.

- Ein applikativer Ausdruck (`f` $\texttt{exp}_1$ ... $\texttt{exp}_n$) wird ausgewertet, indem
  - die Argumentausdrücke $\texttt{exp}_1, \ldots, \texttt{exp}_n$ unmittelbar, d.h. unausgewertet im Rumpf von `f` für die Parameter von `f` eingesetzt werden und anschließend der so entstandene expandierte Ausdruck ausgewertet wird.

Die Wirkung aufgeschobener Argumentauswertung motiviert folgende Bezeichnungsvarianten:

- Namensparameter-, äußerste Auswertung (engl. call-by-name evaluation, outermost evaluation).

Grundthema aller Auswertungsstrategien

...die Organisation des Zusammenspiels von

- Expandieren (von Funktionsaufrufen)
- Simplifizieren (von Ausdrücken verschieden von Funktionsaufrufen)

mit dem Ziel, Ausdrücke so weit zu vereinfachen wie möglich.

Drei Beispiele zur Illustration

1. Arithmetischer Ausdruck:

   ```
   3 * (9+5) ->> ...
   ```

2. Ausdruck mit nichtrekursivem Funktionsaufruf und elementaren bzw. komplexen Argumentausdrücken:

   ```
   simple :: Int -> Int -> Int -> Int
   simple x y z = (x+z) * (y+z)
   ```

   ```
   simple 2 3 4 ->> ...
   ```

3. Ausdruck mit rekursivem Funktionsaufruf:

   ```
   fac :: Integer -> Integer
   fac n = if n == 0 then 1 else n * fac (n-1)
   ```

   ```
   fac 2 ->> ...
   ```

Bsp. 1: Arithmetischer Ausdruck

Viele Simplifikations (S)-Wege – jeder führt zum selben Wert:

S-Weg 1:

```
        3 * (9+5)
(S) ->> 3 * 14
(S) ->> 42
```

S-Weg 2:

```
        3 * (9+5)
(S) ->> 3*9 + 3*5
(S) ->> 27 + 3*5
(S) ->> 27 + 15
(S) ->> 42
```

S-Weg 3:

```
        3 * (9+5)
(S) ->> 3*9 + 3*5
(S) ->> 3*9 + 15
(S) ->> 27 + 15
(S) ->> 42
```

Bsp. 2a): Aufruf mit elementaren Argumenten

```
simple x y z ::  Int -> Int -> Int -> Int
simple x y z = (x + z) * (y + z)
```

Eine Expansion (E), viele Simplifikationen (S) – jeder Weg führt zum selben Wert:

ES-Weg 1:

```
                simple 2 3 4
        (E) ->> (2 + 4) * (3 + 4)
        (S) ->> 6 * (3 + 4)
        (S) ->> 6 * 7
        (S) ->> 42
```

ES-Weg 2:

```
                simple 2 3 4
        (E) ->> (2 + 4) * (3 + 4)
        (S) ->> (2 + 4) * 7
        (S) ->> 6 * 7
        (S) ->> 42
```

ES-Weg 3:

```
                simple 2 3 4
        (E) ->> ...
```

Bsp. 2b): Aufruf mit komplexen Argumenten

```
simple x y z ::  Int -> Int -> Int -> Int
simple x y z = (x + z) * (y + z)
```

ES-Weg 1: Applikative Auswertung

```
          simple 2 3 ((5+7)*9)
    (S) ->> simple 2 3 12*9
    (S) ->> simple 2 3 108
    (E) ->> (2 + 108) * (3 + 108)
    (S) ->> ...
    (S) ->> 12.210
```

ES-Weg 2: Normale Auswertung

```
          simple 2 3 ((5+7)*9)
    (E) ->> (2 + ((5+7)*9)) * ((3 + (5+7)*9))
    (S) ->> ...
    (S) ->> 12.210
```

Beispiel 3: Rekursiver Aufruf

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))

        fac 2
 (E) ->> if 2 == 0 then 1 else (2 * fac (2 - 1))
 (S) ->> if False then 1 else (2 * fac (2 - 1))
 (S) ->> 2 * (fac (2 - 1))
```

Auch hier gibt es die Möglichkeit

- applikativ (d.h., rechnen auf Argumentposition)
- normal (d.h., Aufruf expandieren)

auswertend fortzufahren; wir führen beide Möglichkeiten im Detail aus.

Bsp. 3: Applikative vs. normale Auswertung

Applikativ:

```
              2 * fac (2-1)            – Arg.vereinf. zuerst
(S)    ->>    2 * fac 1
(E)    ->>    2 * (if 1 == 0 then 1
                          else (1 * fac (1-1)))
(2S)   ->>    2 * (1 * fac (1-1))
(S)    ->>    ...in diesem Stil fortfahren.
```

Normal:

```
              2 * fac (2-1)            – Expansion zuerst
(E)    ->>    2 * (if (2-1) == 0 then 1
                else ((2-1) * fac ((2-1)-1)))
(S)    ->>    2 * (if 1 == 0 then 1
                else ((2-1) * fac ((2-1)-1)))
(S)    ->>    2 * (if False then 1
                else ((2-1) * fac ((2-1)-1)))
(2S)   ->>    2 * (1 * fac ((2-1)-1))
(E)    ->>    ...in diesem Stil fortfahren.
```

Bsp. 3: Applikative Auswertung

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
          fac (1+1)        -- Argument wird sofort ausgewertet.
  (S) ->> fac 2            -- Expansion erst nach max. Argumentvereinf.
  (E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
 (2S) ->> 2 * fac (2-1)    -- Arg. wird sofort ausgewertet.
  (S) ->> 2 * fac 1        -- Exp. erst nach max. Argumentvereinf.
  (E) ->> 2 * (if 1 == 0 then 1
                 else (1 * fac (1-1)))
 (2S) ->> 2 * (1 * fac (1-1))  -- Arg. wird sofort ausgewertet.
  (S) ->> 2 * (1 * fac 0)      -- Exp. erst nach max. Arg.vereinf.
  (E) ->> 2 * (1 * (if 0 == 0 then 1
                      else (0 * fac (0-1))))
 (2S) ->> 2 * (1 * 1)
  (S) ->> 2 * 1
  (S) ->> 2
```

⇝ Unverzügliche Argumentauswertung (engl. applicative order evaluation).

Bsp. 3: Normale Auswertung (1)

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

```
         fac (1+1)        – Sofortige Exp., keine vorh. Arg.vereinf.
  (E) ->> if (1+1) == 0 then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> if 2 == 0 then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> if False then 1
             else ((1+1) * fac ((1+1)-1))
  (S) ->> ((1+1) * fac ((1+1)-1))
  (S) ->> (2 * fac ((1+1)-1))                     – Sofortige Exp.
  (E) ->> 2 * (if ((1+1)-1)== 0 then 1
            else (((1+1)-1) * fac (((1+1)-1)-1)))
 (4S) ->> 2 * (((1+1)-1) * fac (((1+1)-1)-1))
```

Bsp. 3: Normale Auswertung (2)

```
(2S) ->> 2 * (1 * fac (((1+1)-1)-1))   – Sofortige Exp.
 (E) ->> 2 * (1 * (if (((1+1)-1)-1) == 0 then 1
   else (((1+1)-1)-1) * fac ((((1+1)-1)-1)-1)))
(4S) ->> 2 * (1 * (if True then 1
   else (((1+1)-1)-1) * fac ((((1+1)-1)-1)-1)))
 (S) ->> 2 * (1 * 1)
 (S) ->> 2 * 1
 (S) ->> 2
```

⇝ Aufgeschobene Argumentauswertung (engl. normal order evaluation).

Kapitel 13.2

Linksapplikative, linksnormale Auswertungsordnung

Linksapplikative, linksnormale

...Auswertungsordnung als wichtige praktische Operationalisierungen

- applikativer
- normaler

Auswertungsordnung.

Applikativ, normal auszuwerten

...beantwortet eine operationell wichtige Frage der Ausdrucksauswertung:

1. Wie ist mit (Funktions-) Argumenten umzugehen?
   $\leadsto$ Ausgewertet oder unausgewertet übergeben?
   - Applikativ (innerst): Ausgewertet übergeben.
   - Normal (äußerst): Unausgewertet übergeben.

Linksapplikativ, linksnormal auszuwerten beantwortet zusätzlich folgende zweite operationell wichtige Frage:

2. Wo ist im Ausdruck auszuwerten?
   $\leadsto$ Links, rechts, halblinks, in der Mitte?
   - Linksapplikativ (linksinnerst): Linkestinnerste Stelle.
   - Linksnormal (linksäußerst): Linkestäußerste Stelle.

```
2*3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib(2*(5+7))
```

Zentrale Frage

Welche praktischen Auswirkungen haben diese Festlegungen (links-) applikativer und (links-) normaler Auswertungsordnung?

...auf den im Terminierungsfall schließlich berechneten Ausdruckswert: Keine.

Hauptresultate (1)

Die Church/Rosser-Theoreme 12.3.1 und 12.3.2 garantieren:

Theorem 13.2.1

Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

Theorem 13.2.2

Wenn irgendeine Auswertungsfolge terminiert, so terminiert auch die (links-) normale Auswertungsreihenfolge.

Informell: Die (links-) normale Auswertungsordnung terminiert am häufigsten.

Hauptresultate (2)

Korollar 13.2.3

Angesetzt auf einen Ausdruck, kann die normale (operationalisiert z.B. als linksnormale) Auswertungsordnung terminieren, die applikative (operationalisiert z.B. als linksapplikative) Auswertungsordnung aber nicht.

...d.h., (links-) applikative und (links-) normale Auswertungsordnung können sich unterscheiden hinsichtlich

- Terminierungsverhalten, d.h. Terminierungshäufigkeit.
- Terminierungsgeschwindigkeit, d.h. Performanz.

Zur Terminierungsgeschwindigkeit

...anhand von Beispielen über den Funktionen squ, infinite-Inc und first:

Die Funktion squ zur Quadrierung einer ganzen Zahl:

```
squ :: Int -> Int
squ n = n * n
```

Die Funktion infinite zum “ewigen” Inkrement:

```
infinite :: Int
infinite = 1 + infinite
```

Die Funktion first zur Projektion auf die erste Komponente eines Paars:

```
first :: (a,b) -> a
first (x,_) = x
```

Auswertung in linksapplikativer Ordnung

Linksapplikative, linkestinnerste (leftmost-innermost (LI)) Auswertung:

```
          ((17+4) + squ (squ (squ (1+1)))) + (2*11)
(LI-S) ->> (21 + squ (squ (squ (1+1)))) + (2*11)
(LI-S) ->> (21 + squ (squ (squ 2))) + (2*11)
(LI-E) ->> (21 + squ (squ (2*2))) + (2*11)
(LI-S) ->> (21 + squ (squ 4)) + (2*11)
(LI-E) ->> (21 + squ (4*4)) + (2*11)
(LI-S) ->> (21 + squ 16) + (2*11)
(LI-E) ->> (21 + 16*16) + (2*11)
(LI-S) ->> (21 + 256) + (2*11)
(LI-S) ->> 277 + (2*11)
(LI-S) ->> 277 + 22
(LI-S) ->> 299
```

Insgesamt: $1 + 7 + 3 = 11$ Schritte

...davon 7 Schritte für `squ (squ (squ (1+1)))`

Auswertung in linksnormaler Ordnung (1)

Linksnormale, linkestäußerste (leftmost-outermost (LO)) Auswertung:

```
             ((17+4) + squ (squ (squ (1+1)))) + (2*11)
(LO-S) ->> (21 + squ (squ (squ (1+1)))) + (2*11)
(LO-E) ->> (21 + squ (squ (1+1)) * squ (squ(1+1))) + (2*11)
(LO-E) ->>
 (21 + ((squ (1+1))*(squ (1+1))) * squ (squ (1+1))) + (2*11)
(LO-E) ->>
 (21 + ((1+1)*(1+1)*sq (1+1)) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (2*(1+1)*squ (1+1)) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (2*2*squ (1+1)) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (4 * squ (1+1)) * squ (squ (1+1))) + (2*11)
(LO-E) ->> (21 + (4*((1+1)*(1+1))) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (4*(2*(1+1))) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (4*(2*2)) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + (4*4) * squ (squ (1+1))) + (2*11)
(LO-S) ->> (21 + 16 * squ (squ (1+1))) + (2*11)
       ->> ...
```

Auswertung in linksnormaler Ordnung (2)

```
        ->> ...
(LO-S) ->> (21 + (16 * 16)) + (2*11)
(LO-S) ->> (21 + 256) + (2*11)
(LO-S) ->> 277 + (2*11)
(LO-S) ->> 277 + 22
(LO-S) ->> 299
```

Insgesamt: $1 + (1{+}10{+}10{+}1) + 3 = 26$ Schritte

...davon 22 Schritte für `squ (squ (squ (1+1)))`

(Links-)applikativ effizienter als (links-)normal?

Nicht immer; betrachte:

```
first (2*21, squ (squ (squ (1+1))))
```

- (Links-) applikativ ausgewertet:

```
                first (2*21, squ (squ (squ (1+1))))
(LI-S) ->> first (42, squ (squ (squ (1+1))))
       ->> ...
(LI-S) ->> first (42, 256)
(LI-E) ->> 42
```

  Insgesamt: 1+7+1=9 Schritte (davon 7 Schritte für den Wert des nicht benötigten zweiten Arguments!)

- (Links-) normal ausgewertet:

```
                first (2*21, squ (squ (squ (1+1))))
(LO-E) ->> 2*21
(LO-S) ->> 42
```

  Insgesamt: 2 Schritte (das nicht benötigte zweite Argument wird nicht ausgewertet!)

Applikative, normale Auswertungsordnung

Das vorige Beispiel illustriert:

- Ergebnisgleichheit: Terminieren (links-) applikative und (links-) normale Auswertungsordnung angewendet auf einen Ausdruck beide, so terminieren sie mit demselben Resultat (siehe Theorem 13.2.1).
- Aber: Schrittzahlungleichheit: (Links-) applikative und (links-) normale Auswertung können bis zur Terminierung (mit gleichem Endresultat) unterschiedlich viele Expansions- und Simplifikationsschritte benötigen.

Damit verbleibt noch zu illustrieren: (Links-) applikative und (links-) normale Auswertung können sich auch im Terminierungsverhalten als solches unterscheiden:

- Applikativ: Nichttermination, kein Resultat: undefiniert.
- Normal: Termination, sehr wohl ein Resultat: definiert.

(Bem.: Die umgekehrte Situation ist nicht möglich (Theorem 13.2.1)!)

Zum Terminierungsverhalten (1)

Betrachte folgendes Beispiel:

```
first (2*21,infinite)
```

In (links-) applikativer Auswertungsordnung:

```
    first (2*21,infinite)
->> first (42,infinite)
->> first (42,1+infinite)
->> first (42,1+(1+infinite))
->> first (42,1+(1+(1+infinite)))
->> ...
->> first (42,1+(1+(1+(...+(1+infinite)...))))
->> ...
```

Insgesamt: Nichtterminierung, kein Resultat: undefiniert!

Zum Terminierungsverhalten (2)

In (links-) normaler Auswertungsordnung:

```
    first (2*21,infinite)
->> 2*21
->> 42
```

Insgesamt: Terminierung, Resultat nach 2 Schritten: definiert!

(Links-) normale Auswertung und Effizienz

Problem: Naive (links-) normale Auswertung führt häufig zur Mehrfachauswertung von Ausdrücken (siehe etwa Beispiel 2b), Weg 2, oder (links-) normale Auswertung des Ausdrucks `squ (squ (squ (1+1)))`.

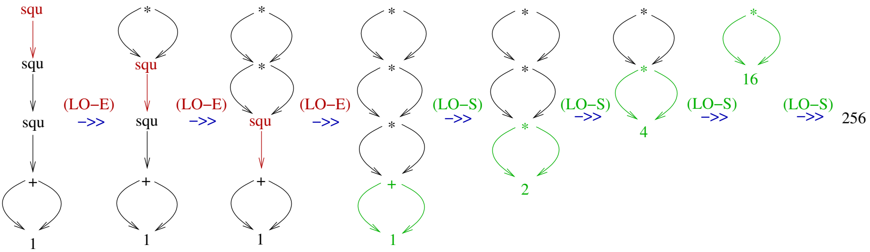
Ziel: Effizienzsteigerung duch Vermeidung unnötiger Mehrfachauswertungen von Ausdrücken.

Methode: Darstellung von Ausdrücken in Form von Graphen, in denen gemeinsame Teilausdrücke geteilt sind; Ausdrucksauswertung in Form von Transformationen dieser Graphen.

Resultat: Verzögerte Auswertung (engl. lazy evaluation)!

...mit der Garantie, dass Argumente höchstens einmal ausgewertet werden (möglicherweise also gar nicht!).

Verzögerte Auswertung: Termrepräsentation, Termtransformation auf Graphen



Insgesamt: 7 Schritte.

(Statt 22 Schritte bei naiver (links-) normaler Auswertung.)

Zusammenfassung

Verzögerte Auswertung (engl. lazy evaluation)

- ist eine effiziente Umsetzung (links-) normaler Auswertungsordnung.
- erfordert implementierungstechnisch eine Darstellung von Ausdrücken in Form von Graphen und Graphtransformationen zu ihrer Auswertung.
- "vergleichbar" performant wie sofortige (engl. eager) Auswertungsordnung, wenn alle Argumente benötigt werden.
- vereint möglichst gut die Vorteile applikativer (Effizienz!) und normaler (Terminierungshäufigkeit!) Auswertungsordnung.

Kapitel 13.3

Auswertungsordnungscharakterisierungen

Charakterisierungen

...von Auswertungsordnungen über Analogien und Betrachtungen zu

(i) Parameterübergabemechanismen
(ii) Auswertungspositionen
(iii) Häufigkeit von Argumentauswertungen
(iv) Definiertheitszusammenhang von Argument und Funktion

(i) Charakterisierung

...über Analogien zu Parameterübergabemechanismen:

- Normale Auswertungsordnung
  - Call-by-name
- Applikative Auswertungsordnung
  - Call-by-value
- Verzögerte Auswertungsordnung
  - Call-by-need

(ii) Charakterisierung

...über die jeweils nächste Auswertungsposition in Ausdrücken:

- Normale Auswertungsordnung
  - Äußerste-Auswertungsordnung: Reduziere nur Redexe, die nicht in anderen Redexen enthalten sind.
  - Linksnormale, Linkestäußerste-Auswertungsordnung: Reduziere stets den linkesten äußersten Redex, der nicht in anderen Redexen enthalten ist.

    Verzögerte (lazy) Auswertungsordnung (effiziente Implementierung linksnormaler Auswertung).
- Applikative Auswertungsordnung
  - Innerste-Auswertungsordnung: Reduziere nur Redexe, die keine Redexe enthalten.
  - Linksapplikative, Linkestinnerste-Auswertungsordnung: Reduziere stets den linkesten innersten Redex, der keine Redexe enthält.

    Sofortige (eager) Auswertungsordnung.

(iii) Charakterisierung

...über die Häufigkeit von Argumentauswertungen:

- Normale Auswertungsordnung
  - Ein Argument wird so oft ausgewertet, wie es benutzt wird.
- Applikative Auswertungsordnung
  - Ein Argument wird genau einmal ausgewertet.
- Verzögerte Auswertungsordnung
  - Ein Argument wird höchstens einmal ausgewertet.

(iii) Veranschaulichung

Betrachte die Funktion:

```
f :: Int -> Int -> Int -> Int
f x y z = if x>42 then y+y else z^z
```

und den Aufruf:

```
f 45 (squ (5*(2+3))) (squ ((2+3)*7))
```

(iii) Veranschaulichung: `f` applikativ ausgew.

```
f x y z = if x>42 then y+y else z^z
```

Applikative Auswertung:

```
           f 45 (squ (5*(2+3))) (squ ((2+3)*7))
(2S) ->> f 45 (squ (5*5)) (squ (5*7))
(2S) ->> f 45 (squ 25) (squ 35)
(2E) ->> f 45 (25*25) (35*35)
(2S) ->> f 45 625 1225
 (E) ->> if 45>42 then 625+625 else 1125^1125
 (S) ->> if True then 625+625 else 1125^1125
 (S) ->> 625+625
 (S) ->> 1250
```

...die Argumente `(squ (5*(2+3)))` und `(squ ((2+3)*7))` werden beide genau einmal ausgewertet (ohne dass der Wert von `(squ ((2+3)*7))` benötigt wird).

(iii) Veranschaulichung: f normal ausgewertet

```
f x y z = if x>42 then y+y else z^z
```

Normale Auswertung:

```
          f 45 (squ (5*(2+3))) (squ ((2+3)*7)))
 (E) ->>  if 45>42 then (squ (5*(2+3))) + (squ (5*(2+3)))
          else (squ ((2+3)*7)) * (squ ((2+3)*7))
 (S) ->>  if True then (squ (5*(2+3))) + (squ (5*(2+3)))
          else (squ ((2+3)*7))^(squ ((2+3)*7))
 (S) ->>  (squ (5*(2+3))) + (squ (5*(2+3)))
(2S) ->>  (squ (5*5)) + (squ (5*5))
(2S) ->>  (squ 25) + (squ 25)
(2E) ->>  (25*25) + (25*25)
(2S) ->>  625 + 625
 (S) ->>  1250
```

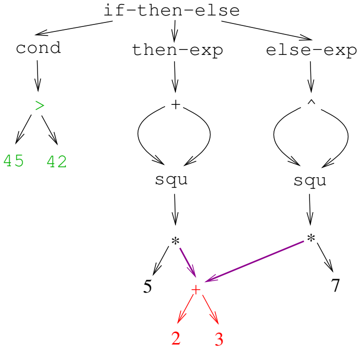
...das Argument `(squ (5*(2+3)))` wird zweimal ausgewertet; das nicht benötigte Argument `(squ ((2+3)*7))` gar nicht.

(iii) Veranschaulichung: `f` verzögert ausgew.

```
f x y z = if 42>x then y+y else z^z
f (5*(2+3)) ... ...
```

f 45 (squ (5*(2+3))) (squ ((2+3)*7))

(E)->>



Verzögerte Auswertung:

(S)->> ... (S)->> 1250

...das Argument `(squ (5*(2+3)))` wird genau einmal ausgewertet; vom nicht benötigten Argument `(squ ((2+3)*7))` der Teilterm `(2+3)` (ohne Extrakosten wg. Ausdrucksteilung!).

(iv) (Teil-) Charakterisierung

...über Definiertheitszusammenhang von Argument und Funktion.

Schüsselbegriff: Striktheit von Funktionen.

Definition 13.3.1 (Strikte Funktionen)

Eine Funktion `f` heißt strikt in einem (bestimmten) Argument `a`, wenn folgendes gilt: Ist der Wert von `a` nicht definiert, so ist auch der Wert von `f` für Aufrufe mit `a` auf dieser Argumentposition nicht definiert.

(iv) Striktheit bei einstelligen Funktionen

Beispiele: Die Fakultäts- und Fibonacci-Funktion sind strikt in ihrem einen Argument. Undefiniertheit des Arguments impliziert Undefiniertheit der Funktion.

```
1 ‘div‘ 0 ->> undef

fac (1 ‘div‘ 0) (lks-appl)->> fac undef (lks-appl)->> undef
fac (1 ‘div‘ 0) (lks-nml)->>
  if ((1 ‘div‘ 0) == 0) then 1
   else n * fac ((1 ‘div‘ 0) - 1) (lks-nml)->> undef

fib (1 ‘div‘ 0) (lks-appl)->> fib undef (lks-appl)->> undef
fib (1 ‘div‘ 0) (lks-nml)->>
 if ((1 ‘div‘ 0) == 0 || (1 ‘div‘ 0) == 1) then 1
  else fib ((1 ‘div‘ 0) - 1) + fib ((1 ‘div‘ 0) - 2)
    (lks-nml)->> undef
```

(iv) Striktheit bei mehrstelligen Funktionen

Mehrstellige Funktionen können

- strikt sein in einigen Argumenten, nicht strikt in anderen.

Beispiel: Der Fallunterscheidungsausdruck (-funktion)

```
(if . then . else .)
```

ist strikt im 1-ten Argument (Bedingung), nicht strikt im 2-ten und 3-ten Argument (then-Ausdruck und else-Ausdruck).

```
if ((1 `div` 0) == 0) then 42 else 1              (strikt in
->> if (undef == 0) then 42 else 1 ->> undef      Bedingung)

if ((0 `div` 1) == 0) then 42 else 1 `div` 0
->> if (0 == 0) then 42 else 1 `div` 0           (nicht strikt
->> if True then 42 else 1 `div` 0 ->> def 42     in 3-tem Arg.)

if ((0 `div` 1) /= 0) then 1 `div` 0 else 42
->> if (0 /= 0) then 1 `div` 0 else 42           (nicht strikt
->> if False then 1 `div` 0 else 42 ->> def 42 in 2-tem Arg.)
```

(iv) Striktheitsinformation und Optimierung

Theorem 13.3.2 (Striktheit und Terminierung)

Für (in einigen Argumenten) strikte Funktionen stimmen (für diese Argumente) das Terminierungsverhalten von sofortiger und verzögerter Auswertungsordnung überein: Durch den Übergang von verzögerter auf sofortige Auswertung (für diese Argumente) gehen keine Ergebnisse verloren (und stimmen nach Theorem 13.2.1 überein).

Bemerkung: Ersetzung von verzögerter durch sofortige Auswertung für strikte Funktionen ist eine der wichtigsten Optimierungen bei der Übersetzung verzögernd auswertender funktionaler Sprachen (siehe Kap. 13.4).

Kapitel 13.4

Sofortige oder verzögerte Auswertung? Eine Standpunktfrage

Frei nach Shakespeare

"To evaluate eagerly or lazily? That is the question."

- Sofortige (engl. eager) Auswertung
  (in Sprachen wie ML, Scheme (ohne Makros),...)
- Verzögerte (engl. lazy) Auswertung
  (in Sprachen wie Haskell, Miranda,...)

...*quot capita, tot sensa — so viele Köpfe, so viele Ansichten.*

Verzögerte vs. sofortige Auswertung (1)

Vorteile verzögerter (lazy) Auswertung:

- Terminiert mit Normalform, wenn es (irgend-) eine terminierende Auswertungsreihenfolge gibt.

  Informell: Verzögerte (und normale) Auswertungsordnung terminieren häufigst möglich!
- Wertet Argumente nur aus, wenn deren Werte wirklich benötigt werden; und dann nur einmal.
- Ermöglicht eleganten und flexiblen Umgang mit potentiell unendlichen Werten von Datenstrukturen (z.B. unendliche Listen, Ströme (siehe Kap. 18.2), unendliche Bäume, etc.).

Verzögerte vs. sofortige Auswertung (2)

Nachteile verzögerter (lazy) Auswertung:

- Konzeptuell und implementierungstechnisch anspruchsvoller.
  - Repräsentation von Ausdrücken in Form von Graphen statt linearer Sequenzen; Ausdrucksmanipulation und -auswertung als Graph- statt Sequenzmanipulation.
  - Partielle Auswertung von Ausdrücken kann Seiteneffekte bewirken! (Beachte: Einwand gilt nicht für Haskell; in Haskell keine Seiteneffekte! In Scheme: Vermeidung von Seiteneffekten obliegt dem Programmierer.)
  - Ein-/Ausgabe nicht in trivialer Weise transparent für den Programmierer zu integrieren.

...volle Einsicht in die Nachteilsursachen erfordert tiefergehendes Verständnis von λ-Kalkül und Bereichstheorie (engl. domain theory).

Verzögerte vs. sofortige Auswertung (3)

Vorteile sofortiger (eager) Auswertung:

- Konzeptuell und implementierungstechnisch einfacher.
- Einfache(re) Integration imperativer Konzepte.
- Vom mathematischen Standpunkt oft "natürlicher".

  Beispiel: Soll der Wert von Ausdrücken wie `(first (2*21,infinite))` definiert gleich 42 sein wie bei verzögerter Auswertung oder undefiniert wg. Nichtterminierung wie bei sofortiger Auswertung?

  first (2*21,infinite)) $\underbrace{\texttt{->>}}_{\texttt{verzögerte Argumentauswertung}}$ 2*21 ->> 42

  first (2*21,infinite))
  $\underbrace{\texttt{->>}}_{\texttt{sofortige Argumentauswertung}}$ first(42,1+infinite)
  ->> first(42,1+(1+infinite)) ->> ...

Auswertungsordnungsauswahlhilfe (1)

... auf Grundlage der Anzahl von Argumentsauswertungen.

Normale Auswertungsordnung (theor. relevant, prakt. nicht)

- Argumente werden so oft ausgewertet, wie sie verwendet werden.
  - \+ Kein Argument wird ausgewertet, dessen Wert nicht benötigt wird.
  - \+ Terminiert, wenn immer es eine terminierende Auswertungsfolge gibt; terminiert am häufigsten, häufiger als applikative Auswertung.
  - – Argumente, die mehrfach verwendet werden, werden auch mehrfach ausgewertet; so oft, wie sie verwendet werden $\rightsquigarrow$ praktisch deshalb irrelevant; praktisch relevant: verzögerte Auswertung.

Auswertungsordnungsauswahlhilfe (2)

Applikative Auswertungsordnung

- Argumente werden genau einmal ausgewertet.
  - \+ Jedes Argument wird exakt einmal ausgewertet; kein zusätzlicher Aufwand über die Auswertung hinaus.
  - – Auch Argumente, deren Wert nicht benötigt wird, werden ausgewertet; das ist kritisch für Argumente, deren Auswertung teuer ist, auf einen Laufzeitfehler führt oder nicht terminiert.

Verzögerte Auswertungsordnung

- Argumente werden höchstens einmal ausgewertet.
  - \+ Ein Argument wird nur ausgewertet, wenn sein Wert benötigt wird; und dann exakt einmal.
  - \+ Kombiniert die Vorteile von applikativer Auswertung (Effizienz!) und normaler Auswertung (Terminierung!).
  - – Erfordert zusätzlichen Aufwand zur Laufzeit für die Verwaltung der Auswertung von (Teil-) Ausdrücken.

Auswertungsordnungsauswahlhilfe (3)

...von pragmatischem Standpunkt aus:

- Applikative, sofortige Auswertungsordnung vorteilhaft gegenüber normaler u. verzögerter Auswertungsordnung, da
  - geringere Laufzeitzusatzkosten (Overhead).
  - größeres Parallelisierungspotential (für Funktionsargumente).
- Verzögerte Auswertungsordnung vorteilhaft gegenüber applikativer, sofortiger Auswertungsordnung, wenn
  - Terminierungshäufigkeit (Definiertheit des Programms!) von überragender Bedeutung.
  - Argumente nicht benötigt (und deshalb gar nicht ausgewertet) werden

    Bsp.: $(\lambda x.\lambda y.y)\,((\lambda x.x\,x)(\lambda x.x\,x))\,z \rightarrow (\lambda y.y)\,z \rightarrow z$
- Als Ideal das Beste beider Welten:
  - Applikativ, sofortig, wo möglich; verzögert, wo nötig.

Auswertungsordnungsauswahlhilfe (4)

...zusammenfassend:

Sofortige (engl. eager) oder verzögerte (engl. lazy) Auswertung:

- Für beide Strategien sprechen gewichtige Gründe.
- Die Wahl ist eine Frage von Standpunkt und Angemessenheit im Anwendungskontext.

Spezialfall: Strikte Funktionen

Für strikte Funktionen stimmen die Terminierungsverhalten von

- sofortiger und verzögerter Auswertungsordnung überein: Durch den Übergang von verzögerter auf sofortige Auswertung gehen keine Ergebnisse verloren (siehe Theorem 13.3.2).
- Die Ergebnisse von verzögerter und sofortiger Auswertung stimmen dabei überein (siehe Theorem 13.2.1).

Striktheit und Optimierung

Für strikte Funktionen darf deshalb stets

- verzögerte durch sofortige Auswertung ersetzt werden, da sich Terminierungsverhalten und Resultat nicht ändern.

Die Ersetzung verzögerter durch sofortige Auswertung ist

- eine der wichtigsten Optimierungstransformationen

von Übersetzern verzögert auswertender funktionaler Sprachen.

Beispiel: Sofortige statt verzögerte Auswertung der in ihrem jeweiligen Argument strikten Fakultäts- und Fibonacci-Funktion.

Striktheitsanalyse, strikte Auswertung

Übersetzer führen deshalb üblicherweise eine

- Striktheitsanalyse

durch, um dort, wo es sicher ist, d.h. wo ein Ausdruck zum Ergebnis beiträgt und deshalb in jeder Auswertungsordnung benötigt wird,

- verzögerte (engl. lazy)

durch

- sofortige (engl. eager)

Auswertung zu ersetzen.

Statt von sofortiger Auswertung spricht man deshalb auch von

- strikter Auswertung (engl. strict evaluation).

Kapitel 13.5

Sofortige und verzögerte Auswertung in Haskell

Steuerung der Auswertung in Haskell

Haskell erlaubt, die Auswertungsordnung (zu einem gewissen Grad) zu steuern.

Verzögerte Auswertung:

- Standardverfahren (vom Programmierer nichts zu tun):

```
        fac (2*(3+5))
(E) ->> if (2*(3+5)) == 0 then 1
        else ((2*(3+5)) * fac ((2*(3+5))-1))
...
```

Sofort(-art)ige Auswertung:

- Mithilfe des zweistelligen Operators ($!):

```
        fac $! (2*(3+5))
(S) ->> fac $! (2*8)
(S) ->> fac $! 16
(E) ->> if 16 == 0 then 1 else (16 * fac (16-1))
...
```

Sofort-artige Auswertung in Haskell (1)

Wirkung des Operators (`$!`):

- Die Auswertung des Ausdrucks `(f $! exp)` erfolgt in gleicher Weise wie die des Ausdrucks `(f exp)` mit dem Unterschied, dass die Auswertung von `exp` erzwungen wird, bevor `f` angewendet und expandiert wird.

Im Detail: Ist `exp` von einem

- elementaren Typ wie `Int`, `Bool`, `Double`, etc., so wird `exp` vollständig ausgewertet.
- Tupeltyp wie `(Int,Bool)`, `(Int,Bool,Double)`, etc., so wird `exp` bis zu einem Tupel von Ausdrücken ausgewertet, aber nicht weiter.
- Listentyp, so wird `exp` so weit ausgewertet, bis als Ausdruck die leere Liste erscheint oder die Konstruktion zweier Ausdrücke zu einer Liste.

Sofort-artige Auswertung in Haskell (2)

In Kombination mit einer curryfizierten Funkion `f` kann

- strikte Auswertung für jede Argumentkombination

erreicht werden.

Beispiel: Für eine zweistellige Funktion `f :: a -> b -> c` erzwingt

- `(f $! x) y`: Auswertung von `x`
- `(f x) $! y`: Auswertung von `y`
- `(f $! x) $! y`: Auswertung von `x` und `y`

vor Anwendung und Expansion von `f`.

Anwendungsbeispiel

Hauptanwendung von ($!) in Haskell: Zur Minderung des Speicherverbrauchs.

Beispiel: Vergleiche

```
lz_sumwith :: Int -> [Int] -> Int
lz_sumwith v []     = v
lz_sumwith v (x:xs) = lz_sumwith (v+x) xs
```

mit

```
ea_sumwith :: Int -> [Int] -> Int
ea_sumwith v []     = v
ea_sumwith v (x:xs) = (ea_sumwith $! (v+x)) xs
```

Anwendungsbeispiel: Verzögerte Auswertung

...liefert:

```
         lz_sumwith 36 [1,2,3]
(E) ->> lz_sumwith (36+1) [2,3,]
(E) ->> lz_sumwith ((36+1)+2) [3]
(E) ->> lz_sumwith (((36+1)+2)+3) []
(E) ->> (((36+1)+2)+3)
(S) ->> ((37+2)+3)
(S) ->> (39+3)
(S) ->> 42                          -- 7 Schritte
```

Anwendungsbeispiel: Sofortige Auswertung

...mittels ($!) liefert:

```
         ea_sumwith 36 [1,2,3]
(E) ->> (ea_sumwith $! (36+1)) [2,3]
(S) ->> (ea_sumwith $! 37) [2,3]
(S) ->> ea_sumwith 37 [2,3]
(E) ->> (ea_sumwith $! (37+2)) [3]
(S) ->> (ea_sumwith $! 39) [3]
(S) ->> ea_sumwith 39 [3]
(E) ->> (ea_sumwith $! (39+3)) []
(S) ->> (ea_sumwith $! 42) []
(S) ->> ea_sumwith 42 []
(E) ->> 42                          -- 10 Schritte
```

Anwendungsbeispiel: Vergleich

- Verzögerte (lazy) Auswertung v. `lz_sumwith 36 [1..3]`
  - baut den Ausdruck `(((5+1)+2)+3)` vollständig auf, bevor die erste Simplifikation ausgeführt wird.
  - Allgemein: `lz_sumwith` baut einen Ausdruck auf, dessen Größe proportional zur Länge der Argumentliste ist.
  - Problem: Programmabbrüche durch Speicherüberläufe schon bei vergleichsweise kleinen Argumenten möglich: `lz_sumwith 5 [1..10000]`
- Sofortige (eager) Auswertung von `ea_sumwith 36 [1..3]`
  - Simplifikationen werden frühestmöglich ausgeführt.
  - Exzessiver Speicherverbrauch (engl. memory leak) wird (in diesem Beispiel) vollständig vermieden.
  - Aber: Die Zahl der Rechenschritte steigt – Besseres Speicherverhalten wird gegen schlechtere Schrittzahl eingetauscht (engl. trade-off).

Zusammenfassung

Der `($!)`-Operator in Haskell ist

- hilfreich und nützlich, das Speicherverhalten von Programmen zu verbessern.

...allerdings kein Königsweg dazu:

- (bereits kleine) Beispiele erfordern eine sorgfältige Untersuchung des Verhaltens verzögerter und sofortiger Auswertung.

Kapitel 13.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (1)

- Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North Holland, 1984. (Kapitel 13, Reduction Strategies)
- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 7.1, Lazy Evaluation)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 7.1, Lazy evaluation)
- Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.2, Models of Reduction; Kapitel 6.3, Reduction Order and Space)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (2)

- Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 2.3, Reduction Strategies)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 2.1, Parameterübergabe und Auswertungsstrategien)
- Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 3, Reduction; Kapitel 8.1, Reduction Machines)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 15, Lazy evaluation; Kapitel 15.2, Evaluation strategies; Kapitel 15.7, Strict application)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (3)

- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 4.4, Applicative Order Reduction; Kapitel 8, Evaluation; Kapitel 8.2, Normal Order; Kapitel 8.3, Applicative Order; Kapitel 8.8, Lazy Evaluation)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 3.1, Reduction Order)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 17.1, Lazy evaluation; Kapitel 17.2, Calculation rules and lazy evaluation)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (4)

- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.
  (Kapitel 17.1, Lazy evaluation; Kapitel 17.2, Calculation rules and lazy evaluation)
- Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.
  (Kapitel 7, Naming; Kapitel 7.1, Parameter Passing)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (5)

- Allen B. Tucker (Editor-in-Chief). *Computer Science. Handbook*. Chapman & Hall/CRC, 2004. (Kapitel 92, Functional Programming Languages (History of Functional Languages, Pure vs. Impure Functional Languages, Nonstrict Functional Languages, Scheme, Standard ML, and Haskell, Research Issues in Functional Programming, etc.))
- Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3.2, A Simple Functional Programming Language – Evaluation Strategies)

Kapitel 14

Typprüfung, Typinferenz

Kapitel 14.1

Motivation

Typisierte Programmiersprachen

...teilen sich in Sprachen mit

- schwacher (Typprüfung zur Laufzeit)
- starker (Typprüfung, -inferenz zur Übersetzungszeit)

Typisierung.

Vorteile

...stark typisierter Programmiersprachen:

- Verlässlicherer Code: Der Nachweis der Typkorrektheit eines Programms ist ein Korrektheitsbeweis für ein Programm auf dem Abstraktionsniveau von Typen. Viele Programmierfehler können so schon zur Übersetzungszeit entdeckt werden.
- Effizienterer Code: Keine Typprüfungen zur Laufzeit nötig.
- Effektivere Programmentwicklung: Typinformation ist Programmdokumentation und vereinfacht Verstehen, Wartung und Weiterentwicklung eines Programms wie z.B. bei der Suche nach vordefinierten Bibliotheksfunktionen: "Gibt es eine Funktion, die alle Duplikate aus einer Liste entfernt" erlaubt (in Haskell), die Suche einzuschränken auf Funktionen mit Typ `((Eq a) => [a] -> [a])`.

Haskell

...ist eine stark typisierte Programmiersprache.

Dabei gilt:

- Jeder gültige Ausdruck hat einen definierten Typ; gültige Ausdrücke heißen wohlgetypt.
- Typen gültiger Ausdrücke können sein:
  - Monomorph
    `fac :: Integer -> Integer`
  - Parametrisch polymorph (uneingeschränkt polymorph)
    `flip :: (a -> b -> c) -> (b -> a -> c)`
  - *Ad hoc* polymorph (eingeschränkt polymorph)
    `elem :: Eq a => a -> [a] -> Bool`
- Typen können angegeben sein:
  - explizit: Typprüfung (grundsätzlich) ausreichend.
  - implizit: Typinferenz erforderlich.

Typprüfung, Typinferenz

...sind Schlüsselfertigkeiten von Übersetzern, Interpretierern.

Betrachte den Ausdruck:

```
magicType = let
              pair x y z = z x y
              f y = pair y y
              g y = f (f y)
              h y = g (g y)
             in h (\x->x)
```

...zur Veranschaulichung der Mächtigkeit automatischer Typinferenzverfahren.

Veranschaulichung (1)

...automatische Typinferenz in Hugs mit dem Kommando :t liefert:

```
Main>:t magicType

magicType ::
((((((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
(((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
(((((((a -> a) -> (a -> a) -> b) -> b) -> (((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> (((((a -> a) ->
(a -> a) -> b) -> b) -> (((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Veranschaulichung (2)

...Klammerebenen farblich hervorgehoben:

```
Main>:t magicType

magicType ::
(((((((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
(((((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
((((((((a -> a) -> (a -> a) -> b) -> b) -> (((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> (((((a -> a) ->
(a -> a) -> b) -> b) -> (((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Veranschaulichung (3)

...Klammerebenen farblich durchgezählt lassen grobe "Strukturen" erkennen:

```
Main>:t magicType

magicType ::
(1
 (1(1(1(1(1(1(a -> a) -> (a -> a) -> b1) -> b   1) ->
  (2(2(a -> a) -> (a -> a) -> b2) -> b2) -> c1) -> c1) ->
  (2(2(3(3(a -> a) -> (a -> a) -> b3) -> b3) ->
  (4(4(a -> a) -> (a -> a) -> b4) -> b4) -> c2) -> c2) ->
  d1) -> d
 1)
 -> (2(2(3(3(5(5(a -> a) -> (a -> a) -> b5) -> b5) ->
      (6(6(a -> a) -> (a -> a) -> b6) -> b6) -> c3) -> c3) ->
      (4(4(7(7(a -> a) ->
      (a -> a) -> b7) -> b7) -> (8(8(a -> a) ->
      (a -> a) -> b8) -> b8) -> c4) -> c4) -> d2) -> d
    2) -> e1) -> e
```

Veranschaulichung (4)

...wobei es auch bleibt.

```
(1
  (1
   (1
    (1
     (1
      (1
       (1(a -> a) -> (a -> a) -> b
       1) -> b
      1) -> (2
             (2(a -> a) -> (a -> a) -> b
             2) -> b
            2) -> c
     1) -> c
    1) -> (2
           (2
            (3
             (3(a -> a) -> (a -> a) -> b
             3) -> b
            3) -> (4
                   (4(a -> a) -> (a -> a) -> b
                   4) -> b
                  4) -> c
           2) -> c
          2) -> d
   1) -> d
  1)
  -> (2(2(3(3(5(5(a -> a) -> (a -> a) -> b5) -> b5) ->
        (6(6(a -> a) -> (a -> a) -> b6) -> b6) -> c3) -> c3) ->
         (4(4(7(7(a -> a) -> (a -> a) -> b7) -> b7) ->
          (8(8(a -> a) -> (a -> a) -> b8) -> b8) -> c4) -> c4) -> d2) -> d2) -> e
1) -> e
```

Automatische Typprüfung, Typinferenz

...der Typ von `magicType` ist fraglos komplex.

Wie gelingt es Übersetzern, Interpretierern, Typen von Ausdrücken wie `magicType` automatisch zu inferieren?

Informell: Durch Auswertung von

- Kontextinformationen in Ausdrücken, Funkionsdefinitionen und Typklassen.

Methoden und Werkzeuge:

- Typanalyse, Typprüfung
- Typsysteme, Typinferenz
- Unifikation

...die wir als nächstes (beispielgetrieben) näher betrachten.

Kapitel 14.2

Monomorphe Typprüfung

Monomorphe Typprüfung

...liefert als Ergebnis: Ein gegebener Ausdruck ist

- wohlgetypt, d.h. hat einen eindeutig bestimmten konkreten Typ.
- nicht wohlgetypt, d.h. hat überhaupt keinen Typ.

Vereinbarung

...für die folgenden Beispiele.

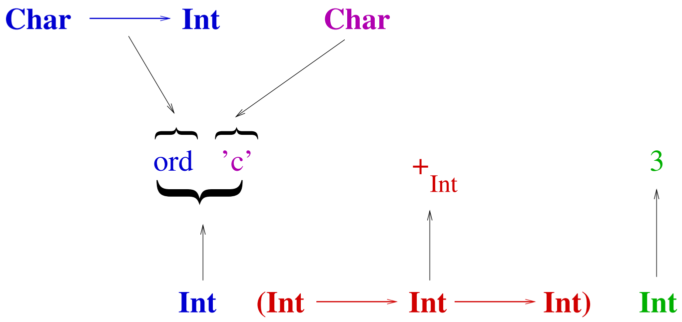
Polymorphie parametrisch oder überladen polymorpher vordefinierter Funktionen in Haskell wird durch geeignete Typindizierung syntaktisch aufgelöst wie nachstehend angedeutet:

- $+_{Int}$ `:: Int -> Int -> Int`
- $*_{Double}$ `:: Double -> Double -> Double`
- $\texttt{length}_{Char}$ `:: [Char] -> Int`
- ...

Typprüfung für Ausdrücke (1)

Beispiel 1: Betrachte den Ausdruck `(ord 'c' +`$_{Int}$` 3)`.

Die Auswertung des Ausdruckskontexts erlaubt

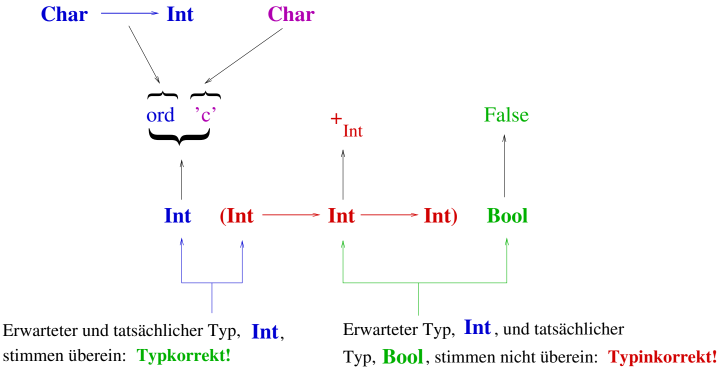


...Typprüfung korrekte Typung festzustellen!

Typprüfung für Ausdrücke (2)

Beispiel 2: Betrachte den Ausdruck `(ord 'c' +`$_{Int}$` False)`.

Die Auswertung des Ausdruckskontexts erlaubt



...Typprüfung inkorrekte Typung aufzudecken!

Typprüfung monomorpher Fkt.-Definitionen

...sei f monomorphe Funktionsdefinition:

```
f :: t1 -> t2 -> ... -> tk -> t
f m1 m2 ... mk
  | w1 = a1
  | w2 = a2
  ...
  | wn = an
```

...für die Kontextauswertung zur Typprüfung für f sind 3 Eigenschaften heranzuziehen:

1. Jeder Wächter w_i muss vom Typ Bool sein.
2. Jeder Ausdruck a_i muss vom Typ t sein.
3. Das Muster jedes Parameters m_i muss konsistent mit dem zugehörigen Typ t_i sein.

Musterkonsistenz, Musterpassung

Informell:

Ein Muster μ ist konsistent mit einem Typ τ, wenn die auf μ passenden Werte vom Typ τ sind.

Detaillierter (vgl. Kapitel 6):

- Eine Variable ist mit jedem Typ konsistent.
- Ein Literal oder Konstante ist mit ihrem Typ konsistent.
- Ein Listenmuster `(p:q)` ist konsistent mit dem Typ `[t]`, wenn `p` mit dem Typ `t` und `q` mit dem Typ `[t]` konsistent ist.
- ...

Beispiele:

- Das Muster `(42:xs)` ist konsistent mit dem Typ `[Int]`.
- Das Muster `(x:xs)` ist konsistent mit jedem Listentyp.

Kapitel 14.3

Polymorphe Typprüfung

Polymorphe Typprüfung

...liefert als Ergebnis: Ein gegebener Ausdruck ist

- wohlgetypt und steht (abkürzend) für einen oder mehrere, möglicherweise unendlich viele konkrete Typen.
- nicht wohlgetypt, d.h. hat überhaupt keinen Typ.

Schlüssel zur algorithmischen Lösung ist das

- Lösen bedingter Systeme von Kontextinformationen (engl. constraint satisfaction)

auf Grundlage der Unifikation von Typausdrücken.

Polymorphe Typprüfung (1)

Beispiel 1: Betrachte die Funktionssignatur von `length` mit dem polymorphen Typ `([a] -> Int)`:

```
length :: [a] -> Int
```

...informell steht `([a] -> Int)` abkürzend für die unendliche Menge konkreter Typen `(`[τ]` -> Int)`, wobei τ Platzhalter für einen beliebigen monomorphen Typ ist:

```
([Int] -> Int)
([(Bool,Char)] -> Int)
([(String -> String)] -> Int)
([(Bool -> Bool -> Bool)] -> Int)
...
```

Polymorphe Typprüfung (2)

In Aufrufkontexten wie

```
length [length [1,2,3], length [True,False,True],
        length [], length [(+),(*),(-)]]
length [(True,'a'), (False,'q'), (True,'o')]
length [reverse, ("Felix" ++), tail, init]
length [(&&), (||), xor, nand, nor]
```

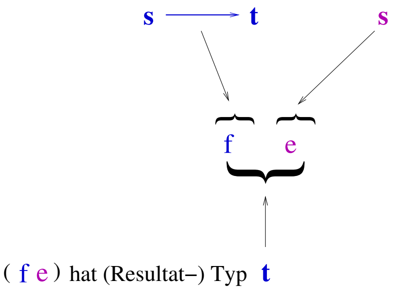
...kann der konkrete monomorphe Typ von Anwendungen von `length` erschlossen werden:

```
length :: [Int] -> Int
length :: [(Bool,Char)] -> Int
length :: [(String -> String)] -> Int
length :: [(Bool -> Bool -> Bool)] -> Int
```

Ausnahme: Der Aufruf (`length []`) erlaubt nur auf `length :: [a] -> Int` zu schließen. Übung: Warum?

Polymorphe Typprüfung (3)

Beispiel 2: Betrachte den applikativen Ausdruck `(f e)`:



Ist weitere Kontextinformation für `f` und `e` nicht vorhanden, liefert die Auswertung des Anwendungskontexts von `(f e)` die allgemeinst möglichen Typen von `e`, `f` und `(f e)` wie folgt:

- `e     :: s`
- `f     :: s -> t`
- `(f e) :: t`

Polymorphe Typprüfung (4)

Beispiel 3: Betrachte die Funktionsgleichung:

```
f (x,y) = (x,['a' .. y])
```

Die Auswertung des Anwendungskontexts ergibt: Funktion `f` erwartet als Argument Paare, an deren

- 1-te Komponente keine Bedingung gestellt ist, die also von einem beliebigen Typ sein darf.
- 2-te Komponente eine Bedingung gestellt ist: `y` muss vom Typ `Char` sein, da `y` als Schranke des Zeichenreihenwerts `['a'..y]` benutzt wird.

Beides zusammen erlaubt den allgemeinsten Typ von `f` zu erschließen:

```
f :: (a,Char) -> (a,[Char])
```

Polymorphe Typprüfung (5)

Beispiel 4: Betrachte die Funktionsgleichung:

```
g (m,zs) = m + length zs
```

Die Auswertung des Anwendungskontexts ergibt: Funktion `g` erwartet als Argument Paare, an deren Komponenten folgende Bedingungen gestellt sind:

- 1-te Komponente: `m` muss von einem numerischen Typ sein, da `m` als Operand von `(+)` verwendet wird.
- 2-te Komponente: `zs` muss vom Typ `[b]` sein, da `zs` als Argument der Funktion `length` verwendet wird, die den Typ `([b] -> Int)` hat.

Beides zusammen erlaubt den allgemeinsten Typ von `g` zu erschließen:

```
g :: (Int, [b]) -> Int
```

Polymorphe Typprüfung (6)

Die Beispiele zeigen, dass wie im monomorphen Fall die Anwendungskontexte von Ausdrücken und Funktionsdefinitionen implizit ein

- System von Typbedingungen festlegen.

Das Typprüfungsproblem reduziert sich so auf die Bestimmung der

- allgemeinst möglichen Typausdrücke, so dass keine Bedingung verletzt ist.

Polymorphe Typprüfung (7)

Beispiel 5: Betrachte die Komposition `(g . f)` mit `f`, `g` aus Bsp. 3 und 4.

In Funktionskompositionen `(h' . h)` ist

- das Resultat der Anwendung von `h` das Argument der Anwendung von `h'`.

Die Auswertung des Anwendungskontexts von `(g . f)` gegeben durch die Gleichungen in Bsp. 3 und 4 ergibt zusätzlich:

- Das Resultat von `f` ist vom Typ `(a,[Char])`.
- Das Argument von `g` ist vom Typ `(Int,[b])`.

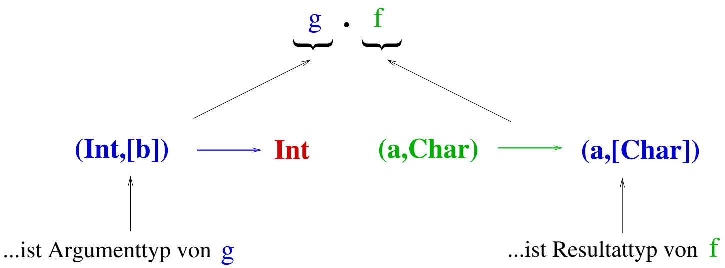
Damit verbleiben noch zu bestimmen: Die

- allgemeinst möglichen Typen für die Typvariablen `a` und `b`, die obige 3 Bedingungen erfüllen.

Der Schlüssel hierfür: Unifikation.

Polymorphe Typprüfung (8)

Veranschaulichung des Unifikationsvorgehens:

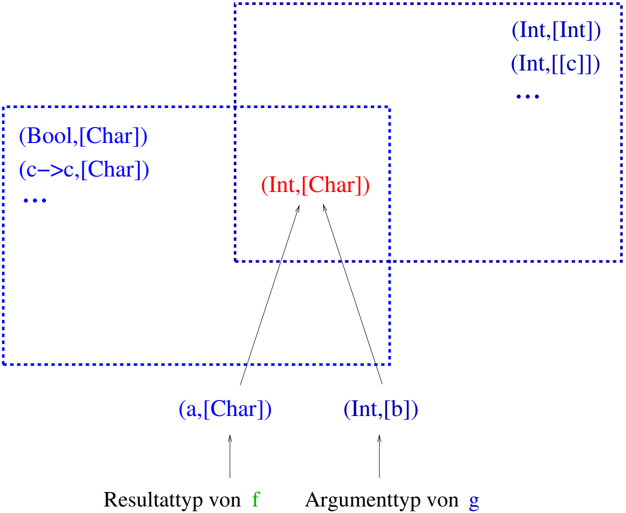


...Unifikation löst die 3 Bedingungen in Kombination auf und liefert `Int` als allgemeinst möglichen Typ für `a`, `Char` für `b` und somit `((Int,Char) -> Int)` für `(g . f)`, d.h.:

```
(g . f) :: (Int,Char) -> Int
```

Polymorphe Typprüfung (9)

...Veranschaulichung des Unifikationsvorgehens:



Instanz, gem. Instanz, Unifikat, Unifikator (1)

Ein Typausdruck a ist (Typ-)

- Instanz eines Typausdrucks a', wenn a aus a' durch konsistentes Ersetzen (oder Substitution) von Typvariablen mit Typausdrücken entsteht.
- gemeinsame Instanz einer Menge M von Typausdrücken, wenn a Instanz von allen Typausdrücken aus M ist.
- allgemeinste gemeinsame Instanz einer Menge M von Typausdrücken, wenn a gemeinsame Instanz von M ist und für alle anderen gemeinsamen Instanzen b von M gilt, dass b Instanz von a ist; a heißt dann (allgemeinstes) Unifikat von M, die zugehörige Substitution (allgemeinster) Unifikator von M.

Instanz, gem. Instanz, Unifikat, Unifikator (2)

...gleichwertig: Ein Typausdruck a ist

- Instanz eines Typausdrucks a', wenn a' sich zu a spezialisieren lässt; wenn a eine Teilmenge von Typen von a' beschreibt.
- gemeinsame Instanz einer Menge M von Typausdrücken, wenn jeder Typausdruck a' aus M sich zu a spezialisieren lässt; wenn a eine Teilmenge von Typen jedes Typausdrucks aus M beschreibt; wenn a eine Teilmenge des Durchschnitts der von den Typausdrücken aus M beschriebenen Typmengen beschreibt.
- allgemeinste gemeinsame Instanz einer Menge M von Typausdrücken, wenn a gemeinsame Instanz von M ist und für alle anderen gemeinsamen Instanzen b von M gilt, dass sich a zu b spezialisieren lässt; dass jede andere gemeinsame Instanz b von M eine Teilmenge der Typen von a beschreibt.

Gemeinsame Instanz v. M $\not\Rightarrow$ Unifikat v. M (1)

Betrachte folgendes Beispiel:

Die Typausdrücke

- `([Bool],[[Bool]])`, `([[e]],[[[e]]])`, `([d],[[d]])`

sind gemeinsame Instanzen (oder Spezialisierungen) der Typausdrücke

- `(a,[a])` und `([b],c)`

unter den Substitutionen

- `[Bool]`, `[[e]]`, `[d]` für `a`
- `Bool`, `[e]`, `d` für `b`
- `[[Bool]]`, `[[[e]]]`, `[[d]]` für `c`.

Gemeinsame Instanz v. M $\not\Rightarrow$ Unifikat v. M (2)

In Substitutionsschreibweise (vgl. Kapitel 12.2):

- `(a,[a])[[Bool]/a]` $=$ `([Bool],[[Bool]])`
  `(a,[a])[[[e]]/a]` $=$ `([[e]],[[[e]]])`
  `(a,[a])[[d]/a]` $=$ `([d],[[d]])`
- `(([b],c)[Bool/b,[[Bool]]/c]` $=$ `([Bool],[[Bool]])`
  `(([b],c)[[e]/b,[[[e]]]/c]` $=$ `([[e]],[[[e]]])`
  `(([b],c)[d/b,[[d]]/c]` $=$ `([d],[[d]])`

Gemeinsame Instanz v. M $\not\Rightarrow$ Unifikat v. M (3)

Weiters sind beide Typausdrücke

- `([Bool],[[Bool]])`, `([[e]],[[[e]]])`

Instanzen (oder Spezialisierungen) des Typausdrucks

- `([d],[[d]])`

unter den Substitutionen `Bool`, `[e]` für `d`:

- `([d],[[d]])[ [Bool]/d]` $=$ `([Bool],[[Bool]])`
  `([d],[[d]])[ [e]/d]` $=$ `([[e]],[[[e]]])`

Umgekehrt ist der Typausdruck

- `([d],[[d]])`

keine Instanz (oder Spezialisierung) von einem der Ausdrücke

- `([Bool],[[Bool]])`, `([[e]],[[[e]]])`.

Gemeinsame Instanz v. M $\not\Rightarrow$ Unifikat v. M (4)

Zusammengefasst:

Die Typausdrücke `([Bool],[[Bool]])`, `([[e]],[[[e]]])`

- sind gemeinsame Instanzen der Typausdrucksmenge

  $M =_{df} \{$`(a,[a])`,`([b],c)`,`([d],[[d]])`$\}$
- jedoch keine allgemeinsten Instanzen von M, d.h. keiner der beiden Ausdrücke ist Unifikat von M.

Der Typausdruck `([d],[[d]])` ist

- die allgemeinste gemeinsame Instanz und damit das Unifikat von M.

Insgesamt ist damit gezeigt:

- Die Eigenschaft "gemeinsame Instanz" einer Menge von Typausdrücken impliziert nicht die Eigenschaft "Unifikat" dieser Menge.

Unifikation, Unifikationsaufgabe

...ist die Bestimmung der

- allgemeinsten gemeinsamen (Typ-) Instanz (engl. most general common (type) instance) einer Menge von Typausdrücken und der zugehörigen Substitution.

Informell: Unifikation

- bestimmt allgemeinstmögliche mehrere Typbedingungen zugleich erfüllende Typausdrücke, die allgemeinste gemeinsame Instanz einer Menge von Typausdrücken sind.
- wertet dafür Kontextbedingungen in Kombination aus.
- führt i.a. zu polymorphen Typausdrücken.
- kann fehlschlagen.

Unifikation bestimmt allgemeinste Instanzen

...unter Auswertung von Kontextbedingungen in Kombination.

Illustriert an Beispiel 5: Unifikation bestimmt unter kombinierter Auswertung der Kontextbedingungenen

```
(g . f)
f (x,y) = (x,['a' .. y])
g (m,zs) = m + length zs
```

den Typausdruck

- (Int,[Char])

als allgemeinste gemeinsame Instanz der Menge M von Typausdrücken

- $M =_{df} \{$(a,[Char]),(Int,[b])$\}$

unter der Substitution Int für a und Char für b:

- (a,[Char])[Int/a] = (Int,[b])[Char/b] = (Int,[Char])

Unifikation liefert polymorphen Typ

Beispiel:

$$
\begin{tikzcd}
(a,[a]) \arrow[dr] & & ([b],c) \arrow[dl] \\
& ([b],[[b]]) &
\end{tikzcd}
$$

([b],[[b]]), die allgemeinste gemeinsame Instanz von (a,[a]) und ([b],c).

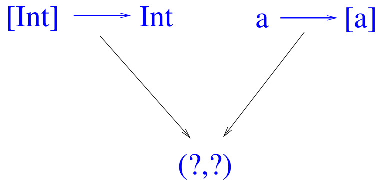
Für die Unifikation von `(a,[a])` und `([b],c)` verlangt die Kontextbedingung

- `(a,[a])`: Die 2-te Komponente ist eine Liste von Elementen des Typs der 1-ten Komponente.
- `([b],c)`: Die 1-te Komponente ist von einem Listentyp.

Zusammen impliziert das: Die allgemeinste gemeinsame Instanz von `(a,[a])` und `([b],c)` ist der (nichtmonomorphe) polymorphe Typausdruck `([b],[[b]])`.

Unifikation schlägt fehl (d.h. ist nicht möglich)

Beispiel:



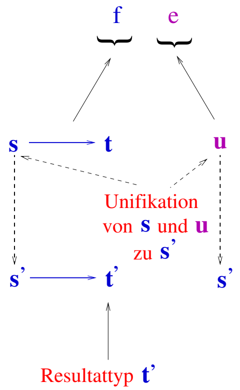
Für die Unifikation von `([Int] -> [Int])` und `(a -> [a])` verlangt die Unifikation der

- Argumenttypen: `a` ist vom Typ `[Int]` ist.
- Resultattypen: `a` ist vom Typ `Int` ist.

Das schließt sich aus und ist nicht zugleich erfüllbar, eine gemeinsame Typinstanz existiert nicht: Unifikation schlägt fehl.

Typüberprüfung von Fkt.-Termen (1)

Betrachte den applikativen Ausdruck `(f e)`:



Es gilt: Typkorrektheit von `(f e)` erfordert nicht Gleichheit von `s` und `u`; es reicht, wenn sie unifizierbar sind: Unifizierter Typ von `f` ist `(s' -> t')`, von `(f e)` somit `t'`.

Typüberprüfung von Fkt.-Termen (2)

Betrachte den applikativen Ausdruck (map ord) mit den Kontextbedingungen:

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unifikation der Typausdrücke (a -> b) und (Char -> Int) liefert als allgemeinst mögliche Typen für (map ord) und map:

```
(map ord) :: [Char] -> [Int]
 map :: (Char -> Int) -> [Char] -> [Int]
```

Typüberprüfung von Fkt.-Termen (3)

Betrachte den applikativen Term `(foldr (+) 0 [3,5,34])` mit den Kontextbedingungen:

```
(foldr (+) 0 [1,2,3,5,7,11,13]) :: Int  (->> 42)
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

Für die Typen von `(foldr (+) 0 [3,5,34])` und `foldr` liefert das:

```
(foldr (+) 0 [1,2,3,5,7,11,13]) :: Int
 foldr :: (Int -> Int - Int) -> Int -> [Int] -> Int
```

Naiv suggeriert dies für den "allgemeinsten" Typ von `foldr`:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

Typüberprüfung von Fkt.-Termen (4)

...eine genauere Überlegung liefert:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Veranschaulichung:

$$\text{foldr}\ \ f\ \ s\ \ [\,]\ \ =\ \ \overset{b}{s}$$

$$\text{foldr}\ \ f\ \ s\ \ (x:xs)\ \ =\ \ \underbrace{f\ \ \overset{a}{x}\ \ \overbrace{(\text{foldr}\ \ f\ \ s\ \ xs\,)}^{b}}_{b}$$

Typprüfung polymorpher Fkt.-Definitionen

...sei f parametrisch polymorphe Funktionsdefinition:

```
f :: t1 -> t2 -> ... -> tk -> t
f m1 m2 ... mk
  | w1 = a1
  | w2 = a2
  ...
  | wn = an
```

...für die Kontextauswertung zur Typprüfung für f sind 3 Eigenschaften heranzuziehen:

1. Jeder Wächter w_i muss vom Typ Bool sein.
2. Jeder Ausdruck a_i muss von einem Typ s_i sein, der mindestens (! – umgekehrt im Aufruffall) so allgemein ist wie der Typ t, d.h. t muss eine Instanz von s_i sein.
3. Das Muster jedes Parameters m_i muss konsistent mit dem zugehörigen Typ t_i sein.

Unifikation mit Konstanten, Variablen (1)

...Konstanten und Variablen werden in Haskell bei Unifikation unterschiedlich behandelt.

Betrachte folgendes Beispiel:

Der Ausdruck a kann erfolgreich getypt werden, die davon abgeleitete Funktionsabstraktion f hingegen nicht:

```
a    = length ([]++[True])
       + length ([]++[1,2,3]) :: Int

f xs = length (xs++[True])
       + length (xs++[1,2,3])  ⇝ Nicht typbar!
```

Unifikation mit Konstanten, Variablen (2)

Das Beispiel zeigt:

- Konstanten wie `[]` können unterschiedlich getypt in Ausdrücken verwendet werden: In `a` verlangt
  - 1-te Verwendung von `[]`: `[] :: [Bool]`
  - 2-te Verwendung von `[]`: `[] :: [Int]`.

  Unifikation analysiert für Konstanten wie `[]` beide Vorkommen getrennt und gelingt.
- Variablen wie `xs` dürfen das nicht: In `f` verlangt
  - 1-te Verwendung von `xs`: `xs :: [Bool]`
  - 2-te Verwendung von `xs`: `xs :: [Int]`.

  Beides zusammen ist unvereinbar. Die verschiedenen Verwendungen von `xs` werden (anders als bei Konstanten) von Unifikation für Variablen nicht getrennt; Unifikation schlägt deshalb für `f` fehl.

Übung 14.3.1

Zeige, dass die unterschiedliche Behandlung von Konstanten und Variablen durch Unifikation sinnvoll ist.

Zu welchen Widersprüchen würde die vermeintlich naheliegende Typisierung

```
f :: [a] -> Int
```

führen? Würde die starke Typisierung von Haskell erhalten bleiben, die zusichert, dass Laufzeitfehler aufgrund von Typfehlern ausgeschlossen sind?

Überlege dazu, Listen welcher Argumenttypen `f` verkraften müsste und ob ihre Implementierung durch die definierende Gleichung

```
f xs = length (xs++[True]) + length (xs++[1,2,3])
```

das hergäbe und was daraus für starke Typisierung und die daraus folgenden Zusicherungen folgte.

Kapitel 14.4

Polymorphe Typprüfung mit Typklassen

Typprüfung mit Typklassen (1)

Betrachte folgende Funktionsdefinition:

```
member []     y = False
member (x:xs) y = (x == y) || member xs y
```

Aus der Auswertung des Kontexts, hier

- dem Listenmuster `(x:xs)` für das erste Argument
- dem Funktionsresultat `False` in der 1-ten Gleichung
- der Benutzung von `(==)` in der 2-ten Gleichung

können wir für den allgemeinsten Typ von `member` schließen:

```
member :: Eq a => [a] -> a -> Bool
```

Typprüfung mit Typklassen (2)

Betrachte zusätzlich zu den definierenden Gleichungen von `member` den Ausdruck `e` mit der Kontextinformation:

```
e :: Ord b => [[b]]
```

Gesucht ist nun der allgemeinste Typ des applikativen Ausdrucks `(member e)`.

Naiv ohne Berücksichtigung der Typklassenkontexte von `e` und `member`, lieferte dies für die Typen von `(member e)`, `member` und `e`:

```
  e :: [[b]]
  member :: [[b]] -> [b] -> Bool
(member e) :: [b] -> Bool
```

Typprüfung mit Typklassen (3)

Mit Berücksichtigung der kombinierten Typklassenkontexte von `member` und `e`:

```
(Eq [b],Ord b)
```

erhalten wir jedoch für den Typ von `(member e)` zunächst:

```
(member e) :: (Eq [b],Ord b) => [b] -> Bool
```

...und schließlich nach einer Typklassenkontextanalyse zur Typklassenkontextvereinfachung einfacher:

```
(member e) :: Ord b => [b] -> Bool
```

Typklassenkontextanalyse (1)

...Analyse und Typklassenkontextvereinfachung erfolgt mehrschrittig, im Bsp. vom Kontext `(Eq [b],Ord b)` zum Kontext `(Ord b)`:

1. Herunterbrechen von Typklassenkontextbedingungen wie `(Eq [b])` auf Bedingungen an Typvariablen wie `b` durch Analyse der involvierten Typklasseninstanzdeklaration wie
   ```
   instance Eq a => Eq [a] where...
   ```
2. Wiederholen von Schritt 1) bis keine Instanzdeklaration mehr anwendbar ist.
3. Weiteres Vereinfachen des Kontexts aus Schritt 2) durch Auswertung der involvierten Typklassendefinitionen wie
   ```
   class Eq a => Ord a where...
   ```
   .

Typklassenkontextanalyse (2)

Für unser Beispiel erhalten wir auf diese Weise:

1. Ausgehend vom Kontext `(Eq [b],Ord b)`, liefert die Analyse der Instanzdeklaration `(instance Eq a => Eq [a] where...)` die Implikation `Eq b`, wenn `Eq [b]`. Das erlaubt `(Eq [b],Ord b)` zu `(Eq b,Ord b)` zu vereinfachen.
2. Keine Instanzdeklaration mehr anwendbar; weiter mit 3).
3. Ausgehend von `(Eq b,Ord b)` aus Schritt 2), liefert die Analyse der Typklassendefinition `(class Eq a => Ord a where...)` die Implikation `Ord b`, wenn `Eq b`. Das erlaubt `(Eq b,Ord b)` zu `(Ord b)` zu vereinfachen; keine weitere Vereinfachung mehr möglich.

Somit erhalten wir ingesamt für den allgemeinsten Typ des applikativen Ausdrucks `(member e)`:

- `(member e) :: Ord b => [b] -> Bool`

Zusammenfassung

...der dreistufige Prozess aus:

- Unifikation
- Analyse (einschl. Instanz- und Typklassendeklarationen)
- Simplifikation

ist das allgemeine Muster für polymorphe Typprüfung mit Typklassen in Haskell.

Kapitel 14.5

Typsysteme, Typinferenz

Typsysteme, Typinferenz

Informell:

Typsysteme sind

- logische Systeme, die uns erlauben, Aussagen der Form "exp ist Ausdruck vom Typ t" zu formalisieren und sie mithilfe von Axiomen und Regeln des Typsystems zu beweisen.

Typinferenz bezeichnet

- den Prozess, den Typ eines Ausdrucks automatisch mithilfe der Axiome und Regeln des Typsystems abzuleiten.

Typgrammatik (typischer Ausschnitt)

...erzeugt eine Typsprache:

$$
\begin{array}{lcll}
\tau & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Char} \mid \mathit{Bool} & \text{(Einfacher Typ)} \\
 & \mid & \alpha & \text{(Typvariable)} \\
 & \mid & \tau \to \tau & \text{(Funktionstyp)} \\
 & & & \\
\sigma & ::= & \tau & \text{(Typ)} \\
 & \mid & \forall\, \alpha.\ \sigma & \text{(Typbindung)}
\end{array}
$$

Sprechweisen: τ ist ein Typ, σ ein Typschema.

Typsystem (typischer Ausschnitt)

...assoziiert mit jedem (typisierbaren) Ausdruck der Sprache einen Typ der Typsprache, wobei Γ eine sogenannte Typannahme (oder Typumgebung) ist:

Axiome:

VAR $$\frac{—}{\Gamma \vdash \mathit{var} : \Gamma(\mathit{var})}$$

CON $$\frac{—}{\Gamma \vdash \mathit{con} : \Gamma(\mathit{con})}$$

Regeln:

COND $$\frac{\Gamma \vdash \mathit{exp} : \mathit{Bool} \quad \Gamma \vdash \mathit{exp}_1 : \tau \quad \Gamma \vdash \mathit{exp}_2 : \tau}{\Gamma \vdash \text{ if } \mathit{exp} \text{ then } \mathit{exp}1 \text{ else } \mathit{exp}2 : \tau}$$

APP $$\frac{\Gamma \vdash \mathit{exp} : \tau' \to \tau \quad \Gamma \vdash \mathit{exp}' : \tau'}{\Gamma \vdash \mathit{exp}\ \mathit{exp}' : \tau}$$

ABS $$\frac{\Gamma[\mathit{var} \mapsto \tau'] \vdash \mathit{exp} : \tau}{\Gamma \vdash \backslash x\text{->}\mathit{exp} : \tau' \to \tau}$$

...

Typumgebung, substituierte Typumgebung

Typumgebungen sind

- partielle Abbildungen, die Typvariablen auf Typschemata abbilden.

Ist Γ eine Typumgebung, so ist $\Gamma[\tau_1/var_1, \ldots, \tau_n/var_n]$

- eine substituierte Typumgebung, die jede Typvariable var_i auf den Typ τ_i abbildet; jede andere Typvariable auf ihren Typ in der Typumgebung Γ.

Unifikationsalgorithmus (schematisch)

$$
\begin{array}{rcl}
\mathcal{U}(\alpha,\alpha) & = & [\,] \\
\mathcal{U}(\alpha,\tau) & = & \left\{\begin{array}{ll} [\tau/\alpha] & \text{falls } \alpha \notin \tau \\ \textit{Fehlschlag} & \text{sonst} \end{array}\right. \\
\mathcal{U}(\tau,\alpha) & = & \mathcal{U}(\alpha,\tau) \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) & = & \mathcal{U}(U\tau_2, U\tau_4)U \text{ mit } U = \mathcal{U}(\tau_1,\tau_3) \\
\mathcal{U}(\tau,\tau') & = & \left\{\begin{array}{ll} [\,] & \text{falls } \tau = \tau' \\ \textit{Fehlschlag} & \text{sonst} \end{array}\right.
\end{array}
$$

Anmerkung:

- Die Anwendung der Gleichungen erfolgt sequentiell von oben nach unten.
- *U* für (allgemeinster) Unifikator (i.w. eine Substitution).

Unifikator, allgemeinster Unifikator

Beispiel: Betrachte die Typausdrücke `(a -> (Bool,c))` und `(Int -> b)`.

Durch scharfes Hinsehen erkennnt man:

Die Substitution `[Int/a,Float/c,(Bool,Float)/b]`

- ist ein Unifikator von `(a -> (Bool,c))`, `(Int -> b)`.

Die Substitution `[Int/a,(Bool,c)/b]`

- ist der (allgemeinste) Unifikator von `(a -> (Bool,c))`, `(Int -> b)`.

Anwendung des Unifikationsalgorithmus

...am Beispiel der Unifikation der Typausdrücke `(a -> c)` und `(b -> Int -> a)`.

Rechnen liefert:

$$\begin{array}{rcl} & & \mathcal{U}(\texttt{a -> c}, \texttt{b -> Int -> a}) \\ (\text{mit } U = \mathcal{U}(\texttt{a},\texttt{b}) = [\texttt{b/a}]) & = & \mathcal{U}(U\texttt{c}, U(\texttt{Int -> a}))U \\ & = & \mathcal{U}(\texttt{c}, \texttt{Int -> b})[b/a] \\ & = & [\texttt{Int -> b/c}][\texttt{b/a}] \\ & = & [\texttt{Int -> b/c}, \texttt{b/a}] \end{array}$$

Damit ist der allgemeinste Unifikator der beiden Typausdrücke die Substitution `[(Int->b)/c,b/a)]` und das (allgemeinste) Unifikat der Typausdrücke `(a -> c)` und `(b -> Int -> a)` der Typausdruck:

- `(b -> Int -> b) =`
  `(a -> c)[(Int->b)/c,b/a)] =`
  `(b -> Int -> a)[(Int->b)/c,b/a)]`

Entscheidend für den Typinferenzalgorithmus

...die syntaxgerichtete Anwendung der Regeln des Typinferenzsystems, d.h. es ist

- stets nur ein Axiom oder eine Regel anwendbar.

Schlüssel dazu: Anpassung des Typinferenzsystems.

Zusammenfassung (1)

Unifikation ist

- zentral für polymorphe Typinferenz.

Das Beispiel der Funktion magicType illustriert die

- Mächtigkeit automatischer Typinferenz.

Das wirft die Frage auf:

- Lohnt es (sich die Mühe anzutun), Typen zu spezifizieren, wenn (auch derart) komplexe Typen wie im Fall von magicType automatisch hergeleitet werden können?

Antwort: ja. Typspezifikationen

- sind eine sinnvolle Kommentierung des Programms.
- ermöglichen Interpretierern und Übersetzern aussagekräftigere Fehlermeldungen zu erzeugen.

Zusammenfassung (2)

Haskell ist stark typisiert:

- Wohltypisierung von Programmen ist deshalb zur Übersetzungszeit entscheidbar. Fehler zur Laufzeit aufgrund von Typfehlern sind deshalb ausgeschlossen.
- Typen können, müssen aber vom Programmierer nicht angegeben werden. Übersetzer und Interpretierer inferieren die Typen von Ausdrücken und Funktionsdefinitionen (in jedem Fall) automatisch.

Zusammenfassung (3)

...Leseempfehlungen zu Typprüfung, Typinferenz.

Für funktionale Sprachen allgemein:

- Anthony J. Field, Peter G. Robinson. Functional Programming. Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)

Spezifisch für Haskell:

- Simon Peyton Jones, John Hughes. Report on the Programming Language Haskell 98. http://www.haskell.org/report/

Zusammenfassung (4)

Überblick Typsysteme:

- John C. Mitchell. Type Systems for Programming Languages. In Jan van Leeuwen (Hrsg.). *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*. Elsevier Science Publishers, 367-458, 1990.

Grundlagen polymorpher Typsysteme:

- Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17:248-375, 1978.
- Luís Damas, Robin Milner. Principal Type Schemes for Functional Programming Languages. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.

Zusammenfassung (5)

Unifikation:

- J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1):23-42, 1965.

Typsysteme, Typinferenz:

- Luca Cardelli. Basic Polymorphic Type Checking. Science of Computer Programming 8:147-172, 1987.

Kapitel 14.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (1)

- Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.
- Luís Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 4.7, Type Inference)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (2)

- Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 6, Type Inference; Kapitel 6.1, Inferring Monomorphic Types; Kapitel 6.2, Polymorphism)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 5, Typisierung und Typinferenz)
- Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (3)

- Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17:248-375, 1978.
- John C. Mitchell. *Type Systems for Programming Languages*. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Jan van Leeuwen (Hrsg.). Elsevier Science Publishers, 367-458, 1990.
- Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. `www.haskell.org/definitions`.
- J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1):23-42, 1965.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (4)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 5, Writing a Library: Working with JSON Data – Type Inference is a Double-Edged Sword)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 13, Checking types)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13, Overloading, type classes and type checking)

Teil VI

Weiterführende Konzepte

Kapitel 15

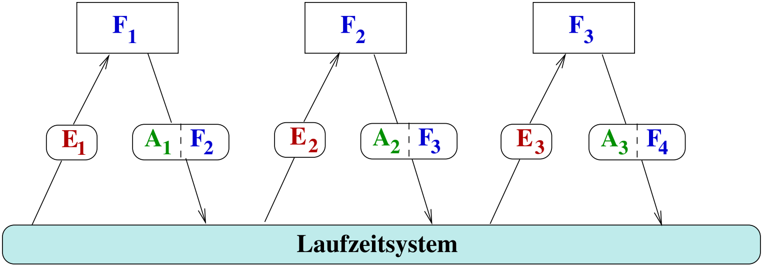
Ein- und Ausgabe

Kapitel 15.1

Motivation

Erwartung

Programme sind dialog- und interaktionsorientiert dank Ein- und Ausgabemöglichkeiten:



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 253.

Aber

Unsere Programme sind bislang stapelverarbeitungsorientiert:

Peter Pepper. *Funktionale Programmierung.*
Springer–Verlag, 2003, S. 245.

Interaktive Ein-/Ausgabebemöglichkeiten fehlen:

- Eingabedaten müssen zu Programmbeginn zur Verfügung gestellt werden, vollständig!

Dialog oder Interaktion zwischen Benutzer und Programm finden nicht statt:

- Einmal gestartet, besteht keine Möglichkeit mehr, mit weiteren Eingaben auf Ergebnisse oder das Verhalten des Programms zu reagieren und es zu beeinflussen.

Kapitel 15.1.1

Die Herausforderung

Die Herausforderung

Konstituierendes Kennzeichen von Lesen und Schreiben, von Ein- und Ausgabe: sie verändern notwendig und irreversibel den Zustand der äußeren Welt.

Ein- und Ausgabe erzeugen

- notwendig und irreversibel Seiteneffekte!

Konstituierendes Kennzeichen rein funktionaler Programmierung:

- Völlige Abwesenheit von Seiteneffekten!

Ein Widerspruch!

Ein-/Ausgabeverzicht ist keine Option

"Der Benutzer lebt in der Zeit
und kann nicht anders als zeitabhängig
sein Programm beobachten."

Peter Pepper. Funktionale Programmierung.
Springer-V., 2. Auflage, 2003.

Das bedeutet insbesondere: Wir dürfen abstrahieren

- von der Arbeitweise des Rechners
- nicht aber von der des Benutzers.

Dialog- und interaktionsorientierte Ein-/Ausgabebehandlung bringt uns an die Nahtstelle

- reiner funktionaler und imperativer Programmierung

und erfordert, sie zu überschreiten.

Kapitel 15.1.2

Warum (naive) Einfachheit versagt

Ein-/Ausgabeoperationen

...in funktionaler Programmierung müssen (wie alle Operationen und Funktionen)

- von funktionalem Typ sein,
- ein Resultat liefern.

Damit zu klären: Was können deren

- Typ und Resultat sein?

Leseoperationen

...liefern stets einen Wert.

- Naheliegend: Den Wert der gelesenen Eingabe.

Am Beispiel einer Leseoperation für ganze Zahlen:

```
-- Zur Illustration: Kein gültiges Haskell!
READ_INT :: INT
READ_INT = << Lies "ganze Zahl"
             -- Der unvermeidbare Seiteneffekt, durch
             -- den der Zustand der Welt irreversibel
             -- verändert wird!
             und liefere deren Wert als Resultat.
             -- Das formal erforderliche und inhalt-
             -- lich auch gewollte Ergebnis der Lese-
             -- operation!
          >>
```

Schreiboperationen

...liefern stets einen Wert.

- Naheliegend: Nichts.
- Hilfsweise: (i) Den geschriebenen Wert, oder (ii) einen Wahrheitswert in Abhängigkeit des Erfolgs der Operation; oder (iii) irgendeinen Wert (beliebig; beliebig, aber fest).

Am Bsp. einer Schreibop. für Zeichen nach (i):

```
-- Zur Illustration: Kein gültiges Haskell!
PRINT_STRING:: STRING -> STRING
PRINT_STRING s =
 << Gib am Bildschirm den Wert von s aus
    -- Der unvermeidbare Seiteneffekt, durch den der
    -- Zustand der Welt irreversibel verändert wird!
    und liefere s als Resultat.
    -- Das formal erforderliche Ergebnis der Schreib-
    -- operation!
 >>
```

Erstes Problem

Betrachte folgende einfache interaktive Programmieraufgabe:

- Schreibe ein Programm, dass (1) eine ganze Zahl liest und anschließend (2) einen frei wählbaren Text schreibt.

Naheliegend: Komponiere die beiden Funktionen READ_INT und PRINT_STRING sequentiell mittels Funktionskomposition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) v = f (g v)
```

Wir erhalten:

```
(PRINT_STRING . READ_INT)
```

Jedoch: Die Komposition scheitert.

```
READ_INT     :: INT
PRINT_STRING :: STRING -> STRING
```

...sind nicht typkompatibel für Komposition mittels (.).

Zweites Problem (1)

Betrachte folgendes Beispiel:

```
konstante = 42 :: INT
fun :: INT -> INT
fun n = n + konstante
fun' :: INT -> INT
fun' n = n + READ_INT
```

- Anders als der Wert von `fun`, hängt der Wert von `fun'` nicht allein vom Argumentwert, sondern auch vom Wert der Eingabeoperation ab.
- Das gilt auch für Funktionen, die sich direkt oder indirekt auf `fun'` abstützen: Die Programmbedeutung wird schwer durchschaubar.
- Jeder Aufruf von `fun'` (u. sich darauf abstützender Funktionen) kann trotz gleichen Arguments einen anderen Wert liefern. Es gilt nicht länger: `fun' 42` == `fun' 42`. Aus Funktionen werden Relationen.

Zweites Problem (2)

Betrachte folgende Wertveinbarungen:

```
wert              = (17+4)*2 :: INT
diff              = wert - wert
diff'             = READ_INT - READ_INT
wahr_oder_falsch = (diff + diff' == 0) :: Bool
```

Ausdruck

- `diff` hat stets den Wert 0; gleich, ob `wert` zunächst als linker oder rechter Operand der Differenz ausgewertet wird.
- `diff'` hat unterschiedliche Werte, wenn die Auswertung von linker und rechter Leseoperation vertauscht wird bei insgesamt gleichen (aber voneinander verschiedenen) eingelesenen Zahlen.
- `wahr_oder_falsch` hat abhängig von `diff'` den Wert `True` oder `False`, ist also nicht konstant.

Damit: Verlust referentieller Transparenz

Die Beispiele zeigen: Ein-/Ausgabe lösen das tragende Grundprinzip reiner funktionaler Programmierung auf:

- Referentielle Transparenz

...und sich daraus ergebende Gewissheiten:

- Keine Veränderungen des Zustands der äußeren Welt (Seiteneffektfreiheit).
- Der Wert eines Ausdrucks hängt nur vom Wert seiner Teilausdrücke ab (Kompositionalität), nicht von der Reihenfolge ihrer Auswertung (Reihenfolgenunabhängigkeit).
- Der Wert eines Ausdrucks ist unveränderlich über die Zeit (Zeitunabhängigkeit); er verändert sich nicht durch die Anzahl seiner Auswertungen (Auswertungshäufigkeitsunabhängigkeit).
- Ein Ausdruck darf stets durch seinen Wert ersetzt werden und umgekehrt (Austauschbarkeit).

Ein-/Ausgabe

...stellt somit ein weiteres leitendes Prinzip reiner funktionaler (und allgemeiner deklarativer) Programmierung infrage:

- Die Betonung des "was" (die Ergebnisse) statt des "wie" (die Art ihrer Berechnung)

...rüttelt insgesamt an den Grundfesten, auf die sich

- reine funktionale Programmierung

gründet und von denen sich ihre

- Stärke und Eleganz

ableiten.

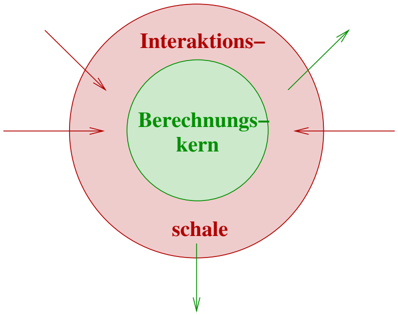
Kapitel 15.2

Haskells Lösung

Haskells E/A-Lösung

Konzeptuell wird in Haskell ein Programm geteilt in

- einen rein funktionalen Berechnungskern
- eine imperativartige Dialog- und Interaktionsschale.



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell.* Pearson, 2004, S. 89.

Haskells Umsetzung

A) Ein neuer (vordefinierter) Datentyp für Ein-/Ausgabe:

- data IO a = ... (Details implementierungsintern versteckt)

Vordefinierte primitive E/A-Operationen:

- ```
  getChar :: IO Char
  getInt  :: IO Int
  ...
  ```
- ```
  putChar :: Char -> IO ()
  putInt  :: Int -> IO ()
  ...
  ```

B) Ein Operator zur Komposition von E/A-Operationen:

- `(>>=) :: IO a -> (a -> IO b) -> IO b`
- 'Syntaktischer Zucker' für `(>>=)`: `do`-Notation.

C) Zwei Vermittlungs'operatoren':

- `return :: a -> IO a`
- "`<-      :: IO a -> a`" ($\rightsquigarrow$ informell!)

Lösungsbeiträge d. Umsetzungsbestandteile (1)

A) Trennung in rein funktionalen Berechnungskern und imperativartige Dialog- und Interaktionsschale:

Der Datentyp (IO a) erlaubt die Unterscheidung von Typen

- des rein funktionalen Berechnungskerns (Char, Int, Bool, etc.)
- der imperativartigen Dialog- und Interaktionsschale ((IO Char)), (IO Int), (IO Bool), etc.)

Effekt: IO-Werte können nicht das gesamte Programm 'kontaminieren'. Vereinbarungen wie für wahr_oder_falsch und fun' sind typinkorrekt und nicht (mehr) möglich; sie werden vom Typsystem ausgeschlossen und abgewiesen:

```
fun' :: Int -> Int        wert = (17+4)*2 :: Int
fun' n = n + getInt       wahr_oder_falsch
                           = (wert - wert) + (getInt - getInt)
```

Lösungsbeiträge d. Umsetzungsbestandteile (2)

B) Festlegung der zeitlichen Abfolge von E/A-Operationen ("Der Benutzer lebt in der Zeit...und kann nicht anders..."):

Der Kompositionsoperator `(>>=)` (oder gleichwertig die `do`-Notation) erlauben die präzise Festlegung der

- zeitlichen Abfolge von E/A-Operationen.

Lösungsbeiträge d. Umsetzungsbestandteile (3)

C) Verbindung von funktionalem Kern und E/A-Schale

- `return`: Von Kern in Schale (in äußere Welt).
- `<-`: Von Schale (von äußerer Welt) in Kern.

Intuitiv: `return` erlaubt rein funktionale Werte (engl. pure values) aus dem funktionalen Kern über die Schale als seiteneffektverursachende Werte (engl. impure values) in die äußere Welt zu transferieren.

`<-` erlaubt den 'reinen' Anteil (`a`-Wert) seiteneffektverursachender Werte (`(IO a)`-Wert) aus der äußeren Welt in den funktionalen Kern zu transferieren.

Bem.: `return` und `<-` verhalten sich in diesem Sinne dual oder invers zueinander, wobei allerdings `return` eine Funktion bezeichnet, `<-` einen Wertvereinbarungsoperator (ähnlich ':=' oder '=') aus imperativen, objektorientierten Sprachen.

Aktionen: Ausdrücke vom Typ (IO a)

Ausdrücke vom Typ (`IO a`)

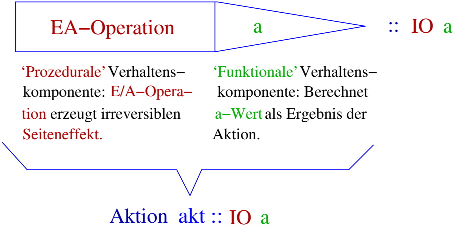
- sind wertliefernde ('funktionaler' Anteil) E/A-Operationen ('prozeduraler' Anteil).
- bewirken einen Lese- oder Schreibseiteneffekt (prozedurales Verhalten) und liefern einen `a`-Wert als Ergebnis (funktionales Verhalten).
- heißen Aktionen (oder Kommandos) (engl. actions oder commands).

Informell:

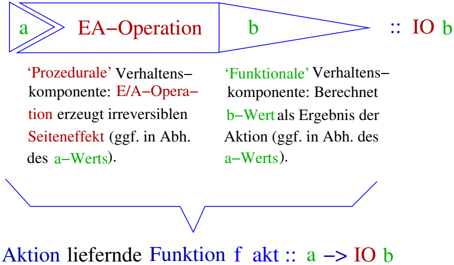
Aktion = (1) E/A-Operation ('prozedural') +
(2) Wertlieferung ('funktional')
= wertliefernde E/A-Operation

Veranschaulichung des Effekts von Aktionen

Aktion `akt :: IO a`



Aktion liefernde Funktion `f_akt :: a -> IO b`



Typ

...aller Leseaktionen ist

- (IO a) (für 'lesegeeignete' Typinstanzen von a).

  Der in einen a-Wert transformierte gelesene Wert wird als (formal erforderliches und inhaltlich gewolltes) Ergebnis von Leseoperationen verwendet.

...aller Schreibaktionen ist

- (IO ()) mit () der einelementige Nulltupeltyp mit gleichbenanntem Datenwert ().

  () als (der einzige) Wert des Nulltupeltyps () wird als formal erforderliches Ergebnis von Schreiboperationen verwendet.

Auswertung, Ausführung von Aktionen

Wegen des kombinierten

- prozeduralen (seiteneffekterzeugende Lese-/Schreiboperation) und
- funktionalen (Wert als Ergebnis liefernden)

Effekts der Auswertung von Aktionen (oder E/A-Ausdrücken), spricht man statt von Auswertung meist von Ausführung von Aktionen (oder E/A-Ausdrücken).

Interpretation der Signatur von (>>=)

...des Kompositionsoperators:

- `(>>=) :: IO a -> (a -> IO b) -> IO b`

Die Signatur liefert:

- `(>>=)` ist eine Abbildung, die eine (Argument-) Aktion mit einem `a`-Wert als Ergebnis (d.h. einen `(IO a)`-Wert) auf eine (Bild-) Aktion mit einem `b`-Wert als Ergebnis abbildet (d.h. auf einen `(IO b)`-Wert) mithilfe einer Funktion, deren Ergebnis angewendet auf den `a`-Ergebniswert der Argumentaktion die gesuchte Bildaktion ist.

Interpretation der Signaturen v. (>>), `return`

...des Kompositionsoperators (>>) und der Funktion `return`:

- `(>>)    :: IO a -> IO b -> IO b`
- `return :: a -> IO a`

Die Signaturen liefern:

- (>>) ist eine Abbildung, die eine (Argument-) Aktion mit einem a-Wert als Ergebnis (d.h. einen (IO a)-Wert) und eine zweite (Argument-) Aktion mit einem b-Wert als Ergebnis (d.h. einen (IO b)-Wert) auf diese zweite Aktion als Bildaktion abbildet.

  (Scheinbar hat das erste Argument keine Bedeutung und verschwindet; dies gilt für sein funktionales Ergebnis, den a-Wert, nicht aber für seinen prozeduralen Lese-/Schreibseiteneffekt!)
- `return` ist eine Abbildung, die einen a-Wert auf eine Aktion mit einem a-Wert als Ergebnis abbildet (d.h. auf einen (IO a)-Wert).

Operationelle Bedeutung

...des Kompositionsoperators:

- `(>>=) :: IO a -> (a -> IO b) -> IO b`

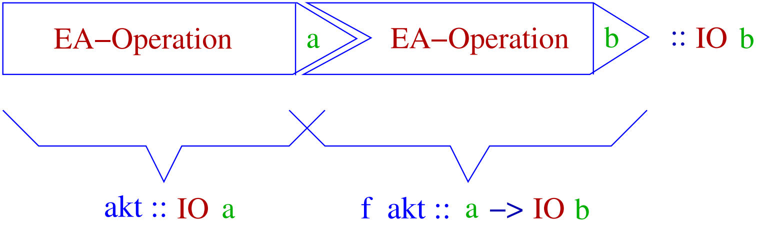
Sei (`akt :: IO a`) eine Aktion, (`f_akt :: a -> IO b`) eine eine Aktion liefernde Abbildung.

Operationelle Bedeutung der Komposition (`akt >>= f_akt`):

- `akt` wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen `a`-Wert; dieser `a`-Wert wird zum Argument von `f_akt`, deren Bildwert vom Typ (`IO b`) eine Aktion ist, die ausgeführt wird, dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen `b`-Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition (`akt >>= f_akt`).

Veranschaulichung

...der operationellen Bedeutung von `(akt >>= f_akt)`:



`akt >>= f_akt` $\hat{=}$ `akt >>= \x -> f_akt x`

Operationelle Bedeutung

...des Kompositionsoperators:

- `(>>) :: IO a -> IO b -> IO b`

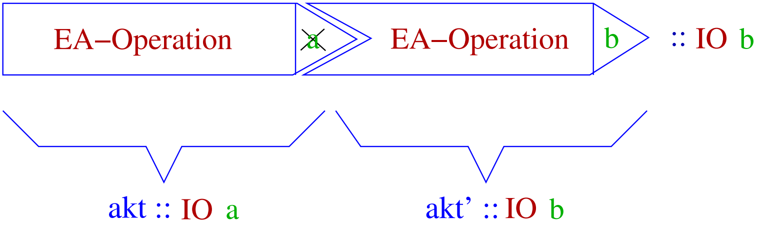
Seien (`akt :: IO a`), (`akt' :: IO b`) Aktionen.

Operationelle Bedeutung der Komposition: (`akt >> akt'`):

- `akt` wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen `a`-Wert. Dieser `a`-Wert wird ignoriert und unmittelbar die Aktion `akt'` ausgeführt, die dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen `b`-Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition (`akt >> akt'`).

Veranschaulichung

...der operationellen Bedeutung von `(akt >> akt')`:



`akt >> akt'` $\widehat{=}$ `akt >> \_ -> akt'`

Operationelle Bedeutung

...der Funktion `return`:

- `return :: a -> IO a`

Sei (`w :: a`) ein `a`-Wert.

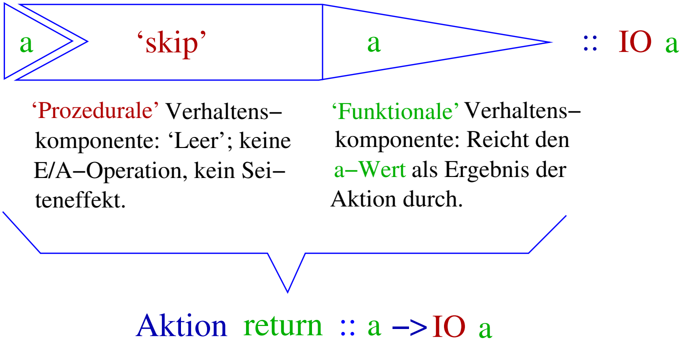
Operationelle Bedeutung von (`return w`):

- `return` bildet `w` in 'offensichtlicher' Weise auf den 'entsprechenden' (`IO a`)-Wert ab, ohne einen Lese- oder Schreibseiteneffekt zu bewirken.

  (Das prozedurale Verhalten von `return` entspricht der leeren Anweisung '*skip*'; `return` hat (deshalb) abweichend von anderen Aktionen nur ein funktionales beobachtbares Verhalten, kein prozedurales).

Veranschaulichung

...der operationellen Bedeutung von `return`:



Beachte

`return` in Haskell

- hat eine gänzlich andere Aufgabe und Bedeutung als das aus imperativen oder objektorientierten Sprachen bekannte *return*; außer der Namensgleichheit besteht weder konzeptuell noch funktionell eine Ähnlichkeit.
- Haskells `return` kann in einer Aktionssequenz auftreten und ausgewertet werden, ohne dass dadurch die Auswertung der restlichen Aktionssequenz beendet würde; `return` kann deshalb auch mehrfach in sinnvoller Weise in einer Aktionssequenz auftreten.
- Zum Verständnis von Haskells `return` ist eine Orientierung am imperativen, objektorientierten *return* deshalb nicht sinnvoll und allenfalls irreführend.

Komposition: 'binde-dann'-, 'dann'-Operator

Die Kompositionsoperatoren

- `(>>=) :: IO a -> (a -> IO b) -> IO b`
- `(>>)  :: IO a -> IO b -> IO b`
  `akt >> akt' = akt >>= \_ -> akt'`   -- vordefiniert

...gelesen als

- binde-dann-Operator (engl. bind oder then)
- dann-Operator (engl. sequence).

Bem.: Die Definition von `(>>)` macht deutlich, dass `(>>)` kein eigenständiger Operator, sondern von `(>>=)` abgeleitet und eine spezielle Anwendung von `(>>=)` ist, die das Ergebnis von `akt` (`a`-Wert) als Argument für `akt'` ignoriert (`\_ -> akt'`: Der `a`-Wert von `akt` wird anders als bei `(>>=)` nicht an einen Namen für weitere Verwendung gebunden, er wird 'vergessen').

Allgemeines Muster von Aktionssequenzen

...mit (>>=) vom Typ (IO b):

```
akt1 >>= \p1 ->                 -- p für Parameter
akt2 >>= \p2 ->
...
aktn >>= \pn ->
(return . f) p1 p2 ... pn
```

mit

```
f  :: a1 ->  a2 -> ...  an ->  b
```

geeigneter Verknüpfungsoperation und

```
akt1 :: IO a1
akt2 :: IO a2
...
aktn :: IO an
```

Aktionssequenzen mit (>>=) und (>>)

...mit und ohne Rückführung von (>>) auf (>>=):

```
akt1 >>= \p1 ->
akt2 >>= \_  ->
akt3 >>= \_  ->
akt4 >>= \p4 ->
...
aktn >>= \pn ->
(return . f) p1 p4 ... pn
```

```
akt1 >>= \p1 ->
akt2 >>
akt3 >>
akt4 >>= \p4 ->
...
aktn >>= \pn ->
(return . f) p1 p4 ... pn
```

...der Typ einer Aktionssequenz ist durch den Typ der letzten Aktion bestimmt.

Schrittweise Aktionssequenzauswertung (1)

```
akt1 >>= \p1 ->
akt2 >>= \p2 ->
akt3 >>= \p3 ->
...
aktn >>= \pn ->
(return . f) p1 p2 p3 ... pn
```

->> (Aktion akt1 erzeugt E/A-Effekt und liefert Wert w1)

```
(\p1 ->
  akt2 >>= \p2 ->
  akt3 >>= \p3 ->
  ...
  aktn >>= \pn ->
  (return . f) p1 p2 p3 ... pn) w1
```

Schrittweise Auswertung Aktionssequenz (2)

```
->>(Aktion akt2 erzeugt E/A-Effekt und liefert Wert w2)

 (\p1 ->
    (\p2 ->
      akt3 >>= \p3 ->
      ...
      aktn >>= \pn ->
      (return . f) p1 p2 p3 ... pn) w1) w2

->> (Aktion akt3 erzeugt E/A-Effekt und liefert Wert w3)
...
->> (Aktion aktn erzeugt E/A-Effekt und liefert Wert wn)

 (\p1 ->
    (\p2 ->
      (\p3 ->
         ...
          (\pn ->
             (return . f) p1 p2 p3 ... pn) w1) w2)...) wn
```

Schrittweise Aktionssequenzauswertung (3)

```
->> (Applikation der 1-ten Funktion auf w1)

    ( \p2 ->
       ( \p3 ->
          ...
           ( \pn ->
             (return . f) w1 p2 p3 ... pn ) w2)...) wn

->> (Applikation der 2-ten Funktion auf w2)

       ( \p3 ->
          ( ...
            ( \pn ->
              (return . f) w1 w2 p3 ... pn ) w3)...) wn

->> (Applikation der 3-ten Funktion auf w3)
...
->> (Applikation der n-ten Funktion auf wn)

              (return . f) w1 w2 w3 ... wn
```

Schrittweise Aktionssequenzauswertung (4)

```
->> (Anwendung der Funktionskompositon '.')
             return (f w1 w2 w3 ... wn)
->> (Anwendung von f auf w1 w2 w3 ... wn liefert w)
             return w
->> (Anwendung von return auf w liefert (ohne E/A-Effekt)
     das Ergebnis erg vom Typ IO b der Aktionssequenz)
             erg :: IO b
```

...in Kapitel 15.4 werden wir Haskells `do`-Notation als suggestivere und bequemere Schreibweise für Aktionssequenzen kennenlernen.

Kapitel 15.2.1

Zur Sonderstellung des Typs `(IO a)`

Zum Unterschied von (IO a) und (MT a) (1)

Vergleiche die Typdeklaration und Wertvereinbarung von:

```
data MT a = MT a                    -- MT für ‘MeinTyp’
bst = MT 'x' :: MT Char      -- bst für ‘buchstabe’
```

mit:

```
data IO a = ...
akt      = getChar :: IO Char
akt'     = putChar 'y' :: IO ()
akt''    = putChar :: Char -> IO ()
akt'''   = return 'z' :: IO Char
akt''''  = return :: a -> IO a
```

Auswertung des Ausdrucks `bst` bewirkt:

- Das Zeichen `'x'` (eingepackt in den Datenwertkonstruktor `MT`) wird geliefert (funktionales Verhalten); darüberhinaus passiert nichts, kein zusätzliches, insbesondere kein prozedurales Verhalten.

Zum Unterschied von (IO a) und (MT a) (2)

Auswertung der Aktion

- akt bewirkt:
  1. (1) Ein Zeichen wird vom Bildschirm gelesen (prozedurale E-Operation) und
  2. (2) der Wert des gelesenen Zeichens wird als Ergebnis (eingepackt in den Datenwertkonstruktor IO) geliefert (funktionales Verhalten).
- akt' bewirkt:
  1. (1) Das Zeichen 'y' wird auf den Bildschirm geschrieben (prozedurale A-Operation) und
  2. (2) der Wert () des Nulltupeltyps () wird als Ergebnis von akt' (eingepackt in den Datenwertkonstruktor IO) geliefert (funktionales Verhalten).
- (akt'' 'y') bewirkt: Ident zu Auswertung von akt'.

Zum Unterschied von (IO a) und (MT a) (3)

Auswertung der Aktion

- `akt'''` bewirkt:
  (1) Ohne dass eine E/A-Operation stattfindet (das prozedurale Verhalten ist ‘leer’, entsprechend ‘*skip*’) wird
  (2) das Zeichen `'z'` als Ergebnis von `akt'''` (eingepackt in den Datenwertkonstruktor `IO`) geliefert (funktionales Verhalten).
- `akt''''` bewirkt: Fehlschlag; `akt'''' = return` vereinbart einen Aliasnamen für die Funktion (genauer: Aktion) `return`. Ohne Argument lassen sich Funktionen nicht auswerten.

  Die Auswertung von z.B. (`akt'''' 'z'`) ist ident mit der von `akt'''`.

Zusammenfassung (1)

Die Ähnlichkeit der Deklarationen

```
data MT a = MT a                   -- rein fkt. Typ
bst =  MT 'x' :: MT Char     -- rein fkt. Ausdruck

data IO a = ...                          -- E/A-Typ
akt = getChar :: IO Char        -- E/A-Op., Aktion
```

ist oberflächlich. Die Auswertung rein funktionaler Ausdrücke wie `bst` ist wesentlich anders als die von E/A-Ausdrücken wie `akt`.

Zusammenfassung (2)

Auswertung eines

- rein funktionalen Ausdrucks (engl. pure expression): Der Wert des Ausdrucks wird geliefert ('funktional'), sonst (passiert) nichts.
- E/A-Ausdrucks (engl. impure expression):
  (1) Eine E/A- Operation wird ausgeführt (Lese-/Schreibseiteneffekt wird generiert, 'prozedural').
  (2) ein a-Wert wird (eingepackt in den Datenwertkonstruktor `IO`) als Ergebnis des E/A-Ausdrucks geliefert ('funktional').

Ausnahme: Der E/A-Ausdruck (`return arg :: IO T`) für (`arg :: T`) und `T` konkreter Typ liefert den Wert von `arg` (eingepackt in den Datenwertkonstruktor `IO`) als Ergebnis ohne eine E/A-Operation auszuführen (und somit ohne Lese-/Schreibseiteneffekt).

Kapitel 15.3

E/A-Operationen, E/A-Sequenzen

Vordefinierte Ein-/Ausgabeoperationen

...zum Lesen und Schreiben vom bzw. auf den Bildschirm.

Eingabeoperationen:

```
getChar :: IO Char
getInt  :: IO Int
getLine :: IO String
...
```

Ausgabeoperationen:

```
putChar :: Char   ->  IO ()
putStr  :: String ->  IO ()
...
```

Vordefinierte Ein-/Ausgabeoperationen

...zum Lesen und Schreiben aus bzw. in Dateien.

Leseoperationen:

```
readFile :: FilePath -> IO String
```

Schreiboperationen:

```
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Dateiende:

```
isEOF :: FilePath -> Bool
```

Pfad- und Dateinamen:

```
type FilePath = String
```

...mit betriebssystemabhängigen Werten von FilePath.

Anwendungen von putStr

...für das "Hallo, Welt"-Programm:

```
halloWelt :: IO ()
putStrLn = putStr "Hallo, Welt!"
```

...zur Ausgabe einer Zeichenreihe mit anschließendem Zeilenumbruch:

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

E/A-Operationen und die Fkt. show, read

Mithilfe der Funktion `show` der Typklasse `Show` und der (globalen (engl. top level)) Funktion `read` (beachte: `read` ist keine Funktion der Typklasse `Read`):

```
show :: Show a => a -> String
```

...lassen sich Werte von Typen der Typklasse `Show` ausgeben und von Typen der Typklasse `Read` einlesen:

```
putLine :: Show a => a -> IO ()
putLine = putStrLn . show

print :: Show a => a -> IO ()
print = putLine

read :: Read a => String -> a
read s = ...                  -- definiert in Prelude
```

Bem.: Vordefinierte Instanzen von `Show` und `Read`: Alle im Prelude definierten Typen mit Ausnahme von Funktions- und `IO`-Typen.

E/A-Sequenzen mittels (.) und (>>=)

...mittels Funktionskomposition (.).

Schreiben mit Zeilenvorschub (vordefinierte Sequenz):

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

..mittels IO-Komposition (>>=).

Lesen einer Zeile und anschließendes Ausgeben der gelesenen Zeile (selbstdefinierte Sequenz):

```
echo :: IO ()
echo = getLine >>= (\zeile -> putLine zeile)
```

Kapitel 15.4

Die do-Notation

Die do-Notation

...'syntaktischer Zucker' für die IO-Kompositionsoperatoren (>>=) und (>>) zur gefälligeren, imperativähnlicheren

- Bildung von Ein-/Ausgabesequenzen.

Beispiel:

```
do zeile <- getLine statt getLine >>= (\zeile ->
   putStrLn zeile                         putStrLn zeile)

do putStr "fun"      statt putStr "fun" >> putStr "\n"
   putStr "\n"       oder  putStr "fun" >>= (\_ ->
                                            putStr "\n")
```

Bemerkung: Ein do-Ausdruck entspricht semantisch einer Sequenz von E/A-Operationen und kann (deshalb) auf eine beliebige Anzahl von Aktionen als Argumente angewendet werden (in den obigen beiden Beispielen jeweils zwei).

Die Abseitsregel gilt auch in do-Ausdrücken.

Allgemeines Muster von do-Ausdrücken

```
do w1 <- akt1       -- Sprechweise: akti Generator
   w2 <- akt2       -- für Wert wi
   ...
   wn <- aktn
   (return . f) w1 w2 ... wn
```

mit

```
f :: a1 -> a2 -> ... -> an -> b
```

geeigneter Verknüpfungsfunktion und

```
akt1 :: IO a1
akt2 :: IO a2
...
aktn :: IO an
```

do-Ausdrücke in einer Zeile

...do-Ausdrücke

```
do w1 <- akt1
   w2 <- akt2
   ...
   wn <- aktn
   (return . f) w1 w2 ... wn
```

lassen sich bedeutungsgleich mit ';' in einer Zeile schreiben (falls gewünscht):

```
do w1 <- akt1; ... ; wn <- aktn;  (return . f) w1 w2 ... wn
```

Der Typ von do-Ausdrücken

...ist durch den Typ der letzten Aktion bestimmt:

```
( do w1 <- akt1
     w2 <- akt2
     ...
     wn <- aktn
     (return . f) w1 w2 ... wn ) :: IO b

f :: a1 -> a2 -> ... -> an -> b
```

bzw.

```
(do w1 <- akt1;...; wn <- aktn; (return . f) w1...wn) :: IO b
```

Nicht verwendete Aktionsergebnisse

...in do-Ausdrücken.

Aktionen liefern stets ein Ergebnis. Bleibt es unverwendet (entspricht Aktionskomposition mit (>>) statt mit (>>=)) kann die Nichtverwendung syntaktisch angedeutet werden, indem ein Aktionsergebnis nicht an wertname, sondern an _ 'gebunden' wird, quasi 'unbenannt' gebunden wird:

```
do w1 <- akt1
   _      <- akt2
   _      <- akt3
   w4 <- akt4
   ...
   wertn <- aktn
   (return . f) w1 w4 .. wn
```

Weglassen unbenannter Bindungen

...noch einfacher können diese unbenannten Bindungen ganz entfallen:

```
do w1 <- akt1
   akt2
   akt3
   w4 <- akt4
   ...
   wn <- aktn
   (return . f) w1 w4 .. wn
```

Entsprechung von do- und (>>=)-Notation

Der do-Ausdruck

```
do w1 <- akt1
   w2 <- akt2
   ...
   wn <- aktn
   (return . f) w1 w2 ... wn
```

entspricht dem (>>=)-Ausdruck und umgekehrt:

```
akt1 >>= \p1 ->
akt2 >>= \p2 ->
...
aktn >>= \pn ->
(return . f) p1 p2 ... pn
```

A-Sequenzen mittels do-Notation

Einmaliges Schreiben einer Zeichenreihe mit Zeilenvorschub:

```
putStrLn :: String -> IO ()                    -- Definition
putStrLn str = do putStr str                  -- aus Prelude
                  putStr "\n"
```

Zweimaliges Schreiben einer Zeichenreihe (mit Z-vorschüben):

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal str = do putStrLn str
                       putStrLn str
```

Viermaliges Schreiben einer Zeichenreihe (mit Z-vorschüben):

```
putStrLn_4mal :: String -> IO ()
putStrLn_4mal str = do putStrLn str
                       putStrLn str
                       putStrLn str
                       putStrLn str
```

E/A-Sequenzen mittels do-Notation

Zwei Lese-, eine Schreibaktion:

```
read2lines_and_report :: IO ()
read2lines_and_report
 = do getLine     -- Z. wird gelesen u. vergessen
      getLine     -- Z. wird gelesen u. vergessen
      putStrLn "Zwei Zeilen gelesen."
```

Eine Lese-, zwei Schreibaktionen:

```
read1line_and_echo2times :: IO ()
read1line_and_echo2times
 = do line <- getLine -- Z. w. gelesen u. gemerkt
      putStrLn line  -- Gemerkte Z. w. ausgegeben
      putStrLn line  -- Gemerkte Z. w. ausgegeben
```

A-Sequenzen parametrisierter Länge

n-maliges Schreiben einer Zeichenreihe (mit Z-vorschüben):

```
putStrLn_Nmal :: Int -> String -> IO ()
putStrLn_Nmal n str
 = if n <= 1
     then putStrLn str
     else do putStrLn str
             putStrLn_Nmal (n-1) str   -- Rekursion!
```

Das erlaubt auch folgende (alternative) Definitionen:

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal = putStrLn_Nmal 2

putStrLn_4mal :: String -> IO ()
putStrLn_4mal = putStrLn_Nmal 4
```

do-Ausdrücke mit return (1)

Lesen einer Zeichenreihe vom Bildschirm und Konversion in eine ganze Zahl:

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

Im Detail:

```
getInt :: IO Int
getInt = do line   <-   getLine
         :: String    :: IO String
            return (read line :: Int)
                 Konvertierung "String" (der
                 Typ von line) zu "Int" (der
                 Argumenttyp von return)
                      :: IO Int
```

do-Ausdrücke mit return (2)

Bestimmung der Länge, der Zeichenzahl einer Datei:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
             name <- getLine
             text <- readFile name
             return (length text)
```

do-Ausdrücke mit return (3)

Im Detail:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
             name    <-    getLine
        :: String      :: IO String
             text    <-    readFile name
        :: String         :: IO String
             return (length text)
                          :: String
                        :: Int
                 :: IO Int
```

Inhalt
Kap. 1
Kap. 2
Kap. 3
Kap. 4
Kap. 5
Kap. 6
Kap. 7
Kap. 8
Kap. 9
Kap. 10
Kap. 11
Kap. 12
Kap. 13
Kap. 14
Kap. 15
15.1
15.2
15.3
15.4
15.5

Dialog- und Interaktionsprogramme

Zwei Frage/Antwort-Interaktionen mit dem Benutzer:

```
ask :: String -> IO String
ask frage = do putStrLn frage
               getLine

interAct :: IO ()          -- Bildschirm-Interaktion
interAct
 = do name <- ask "Sagen Sie mir Ihren Namen?"
      putStrLn ("Willkommen " ++ name ++ "!")

interAct' :: IO ()              -- Datei-Interaktion
interAct'
 = do putStr "Bitte Dateinamen angeben: "
      dateiname <- getLine
      inhalt    <- readFile dateiname
      putStr inhalt
```

Bedeutungsgleichheit von (>>=), (>>) und do

...für die Konstruktion von E/A-Sequenzen.

Die A-Sequenz mittels (>>):

```
writeFile "meineDatei.txt" "Hallo, Dateisystem!"
 >> putStr "Hallo, Welt!"
```

...ist bedeutungsgleich zur A-Sequenz mittels do:

```
do writeFile "meineDatei.txt" "Hallo, Dateisystem!"
   putStr "Hallo, Welt!"
```

Bedeutungsgleichheit von (>>=),(>>) und do

Die E/A-Sequenz mittels (>>=) und (>>):

```
incrementInt :: IO ()
incrementInt
 = getLine >>=
    \zeile -> putStrLn (show (1+read zeile::Int))
```

...ist bedeutungsgleich zur E/A-Sequenz mittels do:

```
incrementInt' :: IO ()
incrementInt'
 = do zeile <- getLine
      putStrLn (show (1 + read zeile :: Int))
```

Informell: 'do' entspricht '(>>=) plus anonyme λ-Abstraktion'.

Bedeutungsgleichheit von (>>=),(>>) und do, return

Die E-Sequenz mittels (>>=):

```
readStringPair :: IO (String,String)
readStringPair
 = getLine >>=
     (\zeile -> (getLine >>=
                  (\zeile' -> (return (zeile,zeile')))))
```

...ist bedeutungsgleich zur E-Sequenz mittels do und return:

```
readStringPair' :: IO (String,String)
readStringPair'
 = do zeile  <- getLine
      zeile' <- getLine
      return (zeile,zeile')
```

Lokale Deklarationen in do-Ausdrücken

Die E/A-Sequenz (ohne lokale Deklarationen):

```
reverse2lines :: IO ()
reverse2lines
 = do line1 <- getLine
      line2 <- getLine
      putStrLn (reverse line2)
      putStrLn (reverse line1)
```

...ist bedeutungsgleich zur Sequenz mit lokalen Deklarationen:

```
reverse2lines :: IO ()
reverse2lines
 = do line1 <- getLine
      line2 <- getLine
      let rev1 = reverse line1
      let rev2 = reverse line2
      putStrLn rev2
      putStrLn rev1
```

Unterschiedliche Bindung von <- und let

Benannte Wertvereinbarungen mittels

- `<-`: für den a-Wert von Aktionen vom Typ `(IO a)` (für 'unreine' Werte aus der äußeren Welt!).
- `let`: für den Wert rein funktionaler Ausdrücke (für 'reine' Werte aus dem rein funktionalen Programmkern).

Rekursive E/A-Programme (1)

...lesen und schreiben gelesener Eingaben: Kopieren.

Nichtterminierendes Kopieren (Abbruch mit `Ctrl-C`):

```
kopiere :: IO ()
kopiere
 = do zeile <- getLine
      putStrLn zeile
      kopiere                          -- Rekursion!
```

n-maliges Kopieren:

```
kopiere_n_mal :: Int -> IO ()
kopiere_n_mal n
 = if n <= 0
    then return ()
    else do zeile <- getLine
            putStrLn zeile
            kopiere_n_mal (n-1)     -- Rekursion!
```

Rekursive E/A-Programme (2)

Kopieren bis zur Eingabe der leeren Zeile:

```
kopiere_bis_leer :: IO ()
kopiere_bis_leer
 = do zeile <- getLine
      if zeile == ""
       then return ()
       else do putStrLn zeile
               kopiere_bis_leer          -- Rekursion!
```

Kopieren bis zur Eingabe der leeren Zeile unter Mitzählen:

```
kopiere_bis_leer_und_zaehle_mit :: Int -> IO ()
kopiere_bis_leer_und_zaehle_mit n
 = do zeile <- getLine
      if zeile == ""
       then putStrLn
             (show n ++ " Zeilen gelesen u. kopiert.")
       else do putStrLn zeile
               kopiere_bis_leer_und_zaehle_mit (n+1)
```

Rekursive E/A-Programme (3)

...summiere eine Folge ganzer Zahlen bis erstmals 0 eingegeben wird:

```
summiere :: IO Int
summiere
 = do n <- getInt
      if n == 0
       then return 0
       else (do m <- summiere
                return (n + m))
```

Vergleiche summiere mit:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns)
 = let m = sum ns
   in (n + m)
```

```
sum’ :: [Int] -> Int
sum’ [] = 0
sum’ (n:ns)
 = n + sum’ ns
```

Rekursive E/A-Programme (4)

...summiere interaktiv eine Folge ganzer Zahlen, bis erstmals 0 eingegeben wird, unter Abstützung auf summiere:

```
summiere_interaktiv :: IO ()
summiere_interaktiv
 = do putStrLn "Gib ganze Zahl ein, je eine pro"
      putStrLn "Zeile. Diese werden summiert bis"
      putStrLn "Null eingegeben wird."
      summe <- summiere
      putStr "Der Summenwert ist "
      putLine summe
```

Iterativartige E/A-Programme

'Iterations'-Ausdruck, -Programm, genauer die iterativartige Funktion `while`:

```
while :: IO Bool -> IO () -> IO ()
while bedingung aktion
  = do b <- bedingung
       if b
        then
         do aktion
            while bedingung aktion  -- Rekursion!
        else
         return ()
```

Zur operationellen Bedeutung der Fkt. `while`

Intuitiv:

- Ist die Bedingung (`bedingung :: IO Bool`) erfüllt (und hat `b` somit den Wert `True`), so wird die Aktion (`aktion :: IO ()`) ausgeführt (`do`-Ausdruck im `then`-Ausdruck); anderenfalls endet die Ausführung/-wertung von `while` ohne weitere E/A-Seiteneffekt mit dem Resultatwert `() :: ()`.
- Nach abgeschlossener Ausführung/-wertung von `aktion` (im Fall der erfüllten Bedingung) wird `while` rekursiv aufgerufen, wodurch insgesamt die 'iterativartige' Anmutung entsteht, dass eine Aktion so lange ausgeführt wird, wie eine Bedingung erfüllt ist.
- Mögliches Argument für die Bedingung: Der Ausdruck

  `isEOF :: IO Bool`

  zum Test auf das Eingabeende.

Anwendung der Funktion `while`

...um eine Datei zeilenweise zu lesen und gelesene Zeilen wieder auszugeben, bis das Dateiende erreicht ist.

```
kopiere_eingabe_nach_ausgabe :: IO ()
kopiere_eingabe_nach_ausgabe
 = while (do wert <- isEOF        -- Arg. f. Param.
             return (not wert))       -- bedingung
         (do zeile <- getLine     -- Arg. f. Param.
             putStrLn zeile)             -- aktion
```

Bem.: Die Klammerung der Argumente von `while` ist nötig.

Wertvereinbarung vs. Wertzuweisung

...genauer: Funktionale Wertvereinbarung vs. imperative Wertzuweisung.

...oder: Zur Natur des

- Wertvereinbarungsoperators '`<-`' in `do`-Ausdrücken

im Vergleich zum

- destruktiven Wertzuweisungsoperator '`:=`' in destruktiven Zuweisung(sanweisung)en (engl. destructive assignments) imperativer Sprachen.

Tatsächlich besitzt

- '`<-`' Ähnlichkeit mit einer Zuweisung, ist aber gänzlich verschieden der destruktiven Zuweisung imperativer Sprachen.

Einmal-Wertvereinbarungsoperator ‘<-’

‘`<-`’ leistet eine Einmal-Wertvereinbarung für einen Namen:

- `zeile <- getLine` bindet das Resultat von `getLine` (allgemeiner: einer Eingabeoperation), an einen Namen, hier `zeile`.
- Diese Verbindung zwischen dem Namen, hier `zeile`, und dem von einer Eingabeoperation gelieferten Wert, hier `getLine`, bleibt für den gesamten Programmlauf erhalten und ist nicht mehr veränderbar.

Mehrfach-Wertzuweisungsoperator ' := '

' := ' leistet eine temporäre Wertzuweisung an eine durch einen Namen bezeichnete Speicherzelle:

- `x := READ_STRING`: Der von `READ_STRING` gelesene Wert wird in die von `x` bezeichnete Speicherzelle geschrieben; der vorher dort gespeicherte Wert wird dabei überschrieben und zerstört (destruktiv!).
- Die durch die Zuweisung geschaffene Verbindung zwischen Namen (d.h. der mit ihm bezeichneten Speicherzelle) und Wert (d.h. dem Inhalt der Speicherzelle) bleibt so lange erhalten (temporär!), bis sie durch eine erneute Zuweisung an diese Zelle überschrieben und zerstört wird (destruktive Zuweisung!).
- Der Inhalt einer Speicherzelle kann jederzeit und beliebig oft überschrieben werden und so die Verbindung von Namen und Wert geändert werden.

Zur Wirkung von Einmal-Wertvereinbarungen

...anhand eines Beispiels:

Aufgabe: Schreibe ein Programm, das so lange eine Zeile vom Bildschirm einliest und wieder ausgibt, bis schließlich die leere Zeile eingelesen wird und die Ausführung abbricht.

Der Effekt von Einmal-Wertvereinbarungen

'Iterativer' Lösungsversuch mittels while-Funktion/Ausdrucks:

```
goUntilEmpty :: IO ()
goUntilEmpty
 = do zeile <- getLine
      while            -- while u. seine Argumente:
       (return (zeile /= []))   -- Bedingungsarg.
       (do putStrLn zeile             -- Aktionsarg.
           zeile <- getLine
           return ())
```

- Die Auswertung von goUntilEmpty terminiert nicht (es sei denn, [] wird als erste Eingabe gewählt).
- zeile und zeile sind unterschiedliche Einmal-Wertvereinbarungen gleichen Namens.
- Test und Ausgabe erfolgen bei jedem Aufruf von while (in jeder 'Schleife') für den Wert von zeile, nie v. zeile.

Lösung: Direkte Rekursion statt ‘Iteration’

Direkt-rekursive Lösung (ohne iterativartigen `while`-Ausdruck):

```
goUntilEmpty' :: IO ()
goUntilEmpty'
 = do zeile <- getLine
      if (zeile == [])
         then return ()
         else (do putStrLn zeile
                  goUntilEmpty')    -- Rekursion!
```

(siehe Simon Thompson. The Craft of Functional Programming. Addison-Wesley/Pearson, 2. Auflage, 1999, S. 393.)

(Subtile) Unterschiede

...in Wertdarstellung und Resultattyp zwischen Ausgabe- und Nichtausgabeoperationen:

```
Main>putStr ('a':('b':('c':[])))
      ->> abc :: IO ()

Main>putChar (head ['a','b','c'])
  ->> a :: IO ()

Main>print "abc"
      ->> "abc" :: IO ()
Main>print 'a'
->> 'a' :: IO ()
```

```
Main>('a':('b':('c':[])))
 ->> "abc" :: [Char]

Main>head ['a','b','c']
 ->> 'a' :: Char

Main>"abc"
 ->> 'a':('b':'c') :: [Char]
Main>'a'
 ->> 'a' :: Char
```

Kapitel 15.5

Zusammenfassung

Haskell-Programme als E/A-Aktionen

Einstiegspunkt für die Auswertung (übersetzter) interaktiver Haskell-Programme ist (per Konvention) eine eindeutig bestimmte

- Definition mit Namen `main` vom Typ `(IO T)`, `T` Typ.
- Intuitiv: "Haskell-Programm = E/A-Aktion".

Beispiel:

```
main :: IO ()                                -- E/A-Schale
main
 = do n <- getInt                            -- E/A-Schale
      let ergebnis = meine_funktion n        -- Fkt. Kern
      putStr ergebnis                        -- E/A-Schale

meine_funktion :: Int -> String              -- Fkt. Kern
meine_funktion n = ... meine_funktion' ...

meine_funktion' :: ...                       -- Fkt. Kern
meine_funktion' ... = ...
...
```

Ein- und Ausgabebehandlung

...in funktionaler und imperativer Programmierung grundsätzlich unterschiedlich. Am augenfälligsten:

- Imperativ: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich.
- Funktional, hier in Haskell: Ein-/Ausgabe an bestimmten Programmstellen konzentriert (in meist wenigen global definierten Funktionen der 'E/A-Schale').

Häufige Beobachtung: Die vermeintliche Einschränkung erweist sich

- als Stärke bei der Programmierung im Großen!

Ein- und Ausgabebehandlung in Haskell

Haskells Konzept zur Behandlung von Ein-/Ausgabe erlaubt Funktionen

- des Berechnungskerns (rein funktionales Verhalten, keine Seiteneffekte)
- der Dialog- und Interaktionsschale (nicht rein funktionales Verhalten, seiteneffektbehaftet).

zu unterscheiden (und konzeptuell zu trennen), kenntlich an den unterschiedlichen Typen, auf deren Werten sie operieren:

`Int`, `Real`, `Char`,... vs. `IO Int`, `IO Real`, `IO Char`,...

mit `IO` vordefinierter Typkonstruktor (wie z.B. `[]`, `(,)`, `(→)`).

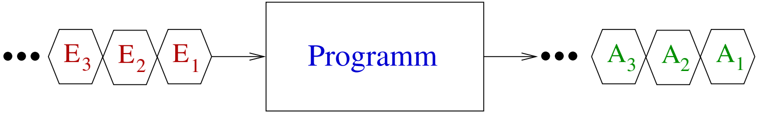
Mithilfe der Kompositionsoperationen `(>>=)` und `(>>)` und der davon abgeleiteten gleichwertigen `do`-Notation ('syntaktischer Zucker') läßt sich die Abfolge von

- Ein-/Ausgabeoperationen präzise festlegen.

Strombasierte Ein-/Ausgabebehandlung (1)

Frühe Haskell-Versionen haben eine strombasierte Behandlung von Ein-/Ausgabe vorgesehen:

- Programme werden dabei als Funktionen auf Strömen angesehen: `IOprog :: String -> String`



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S. 271.

...mit Ein-/Ausgabeströmen für Terminals, Dateisysteme, Drucker, etc.

Strombasierte Ein-/Ausgabebehandlung (2)

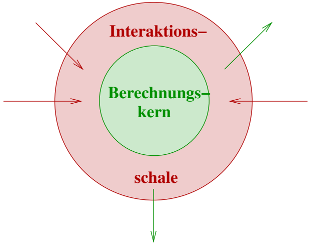
...Vor- und Nachteile für Sprachen mit:

- sofortiger (engl. eager) Auswertung:
  - ein 'echtes' Strommodell fehlt (die Eingabe muss zum Programmstart vollständig vorliegen und konsumiert werden und deshalb endlich sein); Ein-/Ausgabe ist deshalb auf stapelartige (engl. batch-like) Verarbeitung beschränkt.
- verzögerter (engl. lazy) Auswertung:
  - Interaktion ist möglich; verzögerte Auswertung stellt sicher, dass Ein-/Ausgaben in 'richtiger' Abfolge erfolgen.
  - Aber: Ursächlicher und zeitlicher Zusammenhang zwischen Ein-/Ausgaben erscheint oft 'obskur'; besondere Synchronizationen sind nötig, dies zu beheben.
  - Insgesamt: Strombasierte Ein-/Ausgabe kommt an ihre Grenzen beim Übergang zu graphischen Benutzerschnittstellen und wahlfreiem Zugriff auf Dateien.

Haskells heutige Lösung

...der konzeptuellen Trennung eines Haskell-Programms in

- einen rein funktionalen Berechnungskern
- eine Dialog- und Interaktionsschale



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell.* Pearson, 2004, S. 89.

...ist frei von den Problemen strombasierter Ein-/Ausgabebehandlung und wahrt das funktionale Paradigma.

Ausblick (1)

IO ist Typkonstruktor und Instanz der Typ(konstruktor)klasse:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
  m >> k = m >>= \_ -> k      -- Protoimpl. von (>>)
  fail   = error              -- Protoimpl. von fail
```

Vergleiche (mit IO für m):

```
  (>>=)  :: IO a -> (a -> IO b) -> IO b
  (>>)   :: IO a -> IO b -> IO b
  return :: a -> IO a
  fail   :: a -> IO a      -- fail bislang unbenutzt
                           -- von uns.
```

Ausblick (2)

Die Eigenschaften bzw. Anforderungen von Ein-/Ausgabe an funktionale Programmierung und ihre monadische Behandlung in Haskell sind nicht spezifisch, sondern ein Beispiel von vielen, darunter:

- Seiteneffektbehaftete Programmierung
- Nichtdeterminismus
- Fehlerbehandlung
- Programmierung mit großen Datenstrukturen
- ...

Mehr dazu: LVA 185.A05 Fortgeschritte funktionale Programmierung, jeweils im Sommersemester eines Studienjahrs.

Kapitel 15.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 15 (1)

- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 10.1, The IO monad)
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17.5, Ein- und Ausgaben)
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 5.1, IO-Aktionen)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 15 (2)

- Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.5, Input/Output in Functional Programming)
- Andrew J. Gordon. *Functional Programming and Input/Output*. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 16, Communicating with the Outside World)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 15 (3)

- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 10, Interactive programming)
- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 8, Input and output; Kapitel 9, More input and more output)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 21, Ein-/Ausgabe: Konzeptuelle Sicht; Kapitel 22, Ein-/Ausgabe: Die Programmierung)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006. (Kapitel 18, Objekte und Ein-/Ausgabe)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 15 (4)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 7, I/O; Kapitel 9, I/O Case Study: A Library for Searching the Filesystem)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 18, Programming with actions)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 8, Playing the game: I/O in Haskell; Kapitel 18, Programming with monads)
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.

Kapitel 16

Fehlerbehandlung

Typische Fehlersituationen und Sonderfälle

Typische Fehlersituationen:

- Division durch 0.
- Zugriff auf das erste Element einer leeren Liste, `head []`.
- ...

Typische Sonderfälle:

- Auseinanderfallen von intendiertem und implementiertem Definitionsbereichs einer Funktion, z.B.
  - $! : \mathbb{N} \rightarrow \mathbb{N}$: Intendierter Definitionsbereich ist $\mathbb{N}$.
  - `fac :: Integer -> Integer`: Implementierter Definitionsbereich ist $\mathbb{Z}$ (modulo Ressourcenbeschränkungen der Maschine)
- Umgang mit Argumentwerten außerhalb des intendierten Definitionsbereichs.
- ...

Fehlersituationen und Sonderfälle

...bislang von uns naiv behandelt:

Typische Formulierungen aus den Aufgabenstellungen:

...liefert die Funktion den vorher beschriebenen Wert als Resultat; anderenfalls...

- *ist das Ergebnis*
  - *die Zeichenreihe* `"Ungültige Eingabe"`.
  - *die leere Liste* [].
  - *der Wert* 0.
  - ...
- *endet die Berechnung mit dem Aufruf* `error "Ungültige Eingabe"`.
- ...

In diesem Kapitel

...beschreiben wir drei Möglichkeiten eines sukzessive systematisch(er)en Umgangs mit unerwarteten Programmsituationen und Fehlern:

- Panikmodus (Kap. 16.1)
- Vorgabewerte (engl. default values) (Kap. 16.2)
  - Funktionsspezifisch
  - Aufrufspezifisch
- Fehlertypen, Fehlerfunktionen (Kap. 16.3)

Kapitel 16.1

Panikmodus

Panikmodus

Ziel:

- Fehler und Fehlerursache melden, Berechnung stoppen.

Hilfsmittel:

- Die polymorphe Funktion `error :: String -> a`.

Wirkung:

Der Aufruf

- `error "Funktion f: Ungültige Eingabe."`

liefert die Meldung

- `Programmfehler: Funktion f: Ungültige Eingabe.`

und stoppt danach die Programmauswertung unwiderruflich.

Anwendungsbeispiel

Beispiel:

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1
 | n > 0     = n * fac (n-1)
 | otherwise = error "Ungültige Eingabe."

fac 5    ->> 120
fac 0    ->> 1
fac (-5) ->> Programmfehler: Ungültige Eingabe.
```

Beurteilung des Panikmodus

Positiv:

- Schnell und einfach umzusetzen.

Negativ:

- Die Berechnung stoppt unwiderruflich.
- Jegliche Information über den Programmlauf ist verloren, auch sinnvolle.

Kapitel 16.2

Vorgabewerte

Vorgabewerte

Ziel:

- Panikmodus vermeiden; Programmlauf nicht zur Gänze abbrechen, sondern Berechnung möglichst sinnvoll fortführen.

Hilfsmittel: Verwendung von

- funktionsspezifischen (Variante1)
- aufrufspezifischen (Variante 2)

Vorgabewerten (engl. default values) im Fehlerfall.

Variante 1: Funktionsspezifischer Vorgabewert

Vorgabewertvariante 1:

- Im Fehlerfall wird ein funktionsspezifischer Wert als Resultat geliefert.

Beispiel:

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1
 | n > 0     = n * fac (n-1)
 | otherwise = -1
```

Analyse des Beispiels

Im Beispiel der Funktion `fac` gilt:

- Negative Werte treten nie als reguläres Resultat einer Berechnung auf.
- Der funktionsspezifische Vorgabewert -1 erlaubt deshalb, negative Eingaben als fehlerhaft zu erkennen und zu melden, ohne den Programmlauf unwiderruflich abzubrechen.
- Auch `n` selbst käme in diesem Beispiel sinnvoll als Vorgabewert in Frage; die aufrufspezifische Rückmeldung beinhaltete so die ungültige Eingabe selbst, begünstigte dadurch die Fehlersuche und wäre daher sogar aussagekräftiger.

Für beide Vorgabewertwahlen gilt:

- Die Fehlersituation ist für den Programmierer transparent.

Beurteilung der Vorgabewertvariante 1

Positiv

- Panikmodus vermieden, Programmlauf nicht abgebrochen.

Negativ

- Oft gibt es einen zwar naheliegenden und plausiblen funktionsspezifischen Vorgabewert; jedoch kann dieser die Fehlersituation verschleiern und intransparent machen, wenn der Vorgabewert auch als Resultat einer regulären Berechnung auftreten kann.
- Oft fehlt ein naheliegender und plausibler Wert als Vorgabewert; die Wahl eines Vorgabewerts ist in diesen Fällen willkürlich und unintuitiv.
- Oft fehlt ein funktionsspezifischer Vorgabewert gänzlich; Vorgabewertvariante 1 ist in diesen Fällen nicht anwendbar.

...dazu zwei Beispiele.

Vorgabewert vorhanden, aber verschleiernd

Beispiel:

```
rest :: [a] -> [a]
rest (_:xs) = xs
rest []     = []
```

Die Verwendung von [] als funktionsspezifischem Vorgabewert

- liegt nahe und ist plausibel.

Allerdings:

- Das Auftreten der Fehlersituation wird verschleiert und bleibt für den Programmierer intransparent, da [] auch als reguläres Resultat einer Berechnung auftreten kann:

```
rest [42] ->> [] -- [] als reguläres Resultat:
                 -- keine Fehlersituation!
rest []   ->> [] -- [] als irreguläres Resultat:
                 -- Fehlersituation!
```

Kein (naheliegender) Vorgabewert vorhanden

Beispiel:

```
head :: [a] -> a
head (u:_) = u
head []    = ???
```

Ohne Kenntnis der Instanz von `a` ist

- ein `a`-Wert überhaupt nicht angebbar: Vorgabewert fehlt völlig.

Auch mit Kenntnis der Instanz von `a`, z.B., `head :: [Int] -> Int`, bietet sich

- kein `Int`-Wert als Vorgabewert an: Naheliegender, plausibler Vorgabewert fehlt.

...deshalb Übergang zu Vorgabewertvariante 2 mit aufrufspezifischen Vorgabewerten.

Variante 2: Aufrufspezifische Vorgabewerte (1)

Vorgabewertvariante 2:

- Im Fehlerfall wird ein aufrufspezifischer Vorgabewert als Resultat geliefert. Dazu wird die Signatur erweitert und der jeweils gewünschte Vorgabewert als Argument mitgeführt.

Beispiel: Ersetze `head` durch `head'` mit Typisierung

```
head' :: a -> [a] -> a
head' _ (u:_) = u
head' x []    = x
```

...und aufrufspezifischem Fehlerargument `x`.

Variante 2: Aufrufspezifische Vorgabewerte (2)

Generelle Vorgehensweise:

- Ergänze die fehlerbehandlungsfreie Implementierung einer (hier einstellig angenommenen) Funktion `f`:

  ```
  f :: a -> b
  f u = ...
  ```

  um eine fehlerbehandelnde Variante `f'` dieser Funktion:

  ```
  f' :: b -> a -> b
  f' x u
    | fehlerFall = x
    | otherwise  = f u
  ```

  wobei `fehlerFall` die Fehlersituation charakterisiert.

Bemerkung: Im Beispiel der Funktion `head'` konnte die Abstützung auf `head` gemäß der generellen Vorgehensweise umgangen werden.

Beurteilung der Vorgabewertvariante 2 (1)

Positiv:

- Panikmodus vermieden, Programmlauf nicht abgebrochen.
- Generalität, stets anwendbar.
- Flexibilität, aufrufspezifische Vorgabewerte ermöglichen variierende Fehlerwerte und Fehlerbehandlung.

Beurteilung der Vorgabewertvariante 2 (2)

Negativ:

- Transparente Fehlerbehandlung ist nicht gewährleistet, wenn aufrufspezifische Vorgabewerte auch reguläres Resultat einer Berechnung sein können, z.B.:

```
head 'F' "Fehler" ->> 'F' -- reguläres Ergebnis
head 'F' ""       ->> 'F' -- irreguläres Ergebnis
```

- In diesen Fällen Gefahr ausbleibender Fehlerwahrnehmung mit (möglicherweise fatalen) Folgen durch
  - Vortäuschen eines regulären und korrekten Berechnungsablaufs und eines regulären und korrekten Ergebnisses!

    (Typischer Fall eines "sich ein 'x' für ein 'u' vormachen zu lassen!")

Kapitel 16.3

Fehlertypen, Fehlerfunktionen

Erkennen, melden, behandeln von Fehlern (1)

Ziel:

- Systematisches Erkennen, Melden und Behandeln von Fehlersituationen.

Hilfsmittel:

- Dezidierte Fehlertypen, Fehlerwerte und Fehlerfunktionen statt schlichter Vorgabewerte.

Erkennen, melden, behandeln von Fehlern (2)

Zentral:

Meldbarkeit von Fehlern:

- Der (Fehler-) Datentyp

```
data Maybe a = Just a
             | Nothing
             deriving (Eq, Ord, Read, Show)
```

...die Werte des Typs `a` in der Form `Just a` mit dem Zusatzwert `Nothing` als Fehlerwert.

Erkennen, weiterreichen, fangen und behandeln von Fehlern:

- Die Funktionen
  - mapMaybe: Erkennen und weiterreichen von Fehlern.
  - maybe: Fangen und behandeln von Fehlern.

Erkennen und melden von Fehlern (1)

Generelle Vorgehensweise:

- Ergänze die fehlerbehandlungsfreie Implementierung einer (hier einstellig angenommenen) Funktion `f`:

  ```
  f :: a -> b
  f u = ...
  ```

  um die fehlererkennende und -meldende Variante `f'`:

  ```
  f' :: a -> Maybe b
  f' u
   | fehlerFall = Nothing
   | otherwise  = Just (f u)
  ```

  wobei `fehlerFall` die Fehlersituation charakterisiert.

Erkennen und melden von Fehlern (2)

Anwendungsbeispiel:

Ergänze die (vordefinierte) nichtfehlerbehandelnde Funktion `div` um die fehlererkennende und -meldende Variante `div'`:

```
div' :: Int -> Int -> Maybe Int
div' n m
  | (m == 0)  = Nothing
  | otherwise = Just (div n m)
```

Weiterreichen und behandeln von Fehlern

Anders als die Funktion `div`, deren Auswertung im Fehlerfall (d.h., Division durch 0)

- gemäß des Panikmodus

vom Laufzeitsystem abgebrochen wird, ist die Funktion `div'` in der Lage, einen Fehler ohne Auswertungsabbruch

- zu erkennen (`(m == 0)`)
- in Gestalt des Resultats zu melden (`Nothing`).

Offen bleibt:

- Was machen wir im Fehlerfall mit dem Resultat `Nothing`?

Dazu die Funktionen `mapMaybe` und `maybe`...

Die Funktionen mapMaybe und maybe (1)

...erlauben im Zusammenspiel das Erkennen, Weiterreichen, Fangen u. schließliche Behandeln von Fehlern zu organisieren:

Die Funktion mapMaybe:

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just u) = Just (f u)
```

Curryfizierte und uncurryfizierte Sicht auf mapMaybe:

- Curryfiziert: mapMaybe bildet eine (nicht fehlerbehandelnde) Funktion vom Typ (a -> b) auf eine Funktion vom Typ (Maybe a -> Maybe b) ab (entspricht einem "Typ-lifting").
- Uncurryfiziert: mapMaybe bildet einen (Maybe a)-Wert auf einen (Maybe b)-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ (a -> b).

Die Funktionen mapMaybe und maybe (2)

Die Funktion maybe:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f Nothing  = x
maybe x f (Just u) = f u
```

Curryfizierte und uncurryfizierte Sicht auf mapMaybe:

- Curryfiziert: Gegeben einen b-Wert bildet maybe eine Funktion vom Typ (a -> b) auf eine Funktion vom Typ (Maybe a -> b) ab (entspricht einem "Typ-lifting").
- Uncurryfiziert: maybe bildet einen (Maybe a)-Wert auf einen b-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ (a -> b) und eines aufrufspezifischen Fehlerarguments vom Typ b (entspricht der Vorgabewertvariante 2).

Die Funktionen mapMaybe und maybe (3)

...erlauben Fehlerwerte

- weiterzureichen, die Fähigkeit von mapMaybe:

```
mapMaybe f Nothing = Nothing   -- Der Fehlerwert
                               -- Nothing wird
                               -- von mapMaybe
                               -- durchgereicht.
```

- zu fangen und (im Sinn von Vorgabewertvariante 2) zu behandeln, die Fähigkeit von maybe:

```
maybe x f Nothing = x   -- Der aufrufspezifische
                        -- Vorgabewert x wird als
                        -- Resultat geliefert
                        -- (Vorgabewertvariante 2).
```

Anwendungsbeispiel

Zusammenspiel der Funktionen `mapMaybe` und `maybe`:

- Fehlerfall: Der Fehler wird von `mapMaybe` erkannt und später von `maybe` gefangen und behandelt.

```
maybe 9999 (+1) (mapMaybe (*3) (div' 9 0))
->> maybe 9999 (+1) (mapMaybe (*3) Nothing)
->> maybe 9999 (+1) Nothing
->> 9999
```

- Fehlerfreier Fall: Alles läuft "normal" ab.

```
maybe 9999 (+15) (mapMaybe (*3) (div' 9 1))
->> maybe 9999 (+15) (mapMaybe (*3) (Just 9))
->> maybe 9999 (+15) (Just 27)
->> (+15) 27
->> 27 + 15
->> 42
```

Bewertung der Fehlerbehandl'g mittels `Maybe`

Positiv:

- Fehler können erkannt, gemeldet, weitergereicht und schließlich gefangen und (im Sinn von Vorgabewertvariante 2) behandelt werden.

Negativ:

- Geänderte Funktionalität: `Maybe b` statt `b`.

Pragmatischer Zusatzvorteil:

- Systementwicklung ist ohne explizite Fehlerbehandlung möglich (z.B. mit nichtfehlerbehandelnden Funktionen wie `div`).
- Fehlerbehandlung kann nach Abschluss durch Ergänzung der fehlerbehandelnden Funktionsvarianten (wie z.B. der Funktion `div'`) zusammen mit den Funktionen `mapMaybe` und `maybe` realisiert werden.

Kapitel 16.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 16

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 19, Error Handling)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 14.4, Case study: program errors)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14.4, Modelling program errors)

Kapitel 17

Module

Modularisierung

...wichtiges programmiersprachliches Hilfsmittel zur Dekomposition und Strukturierung von Programm(system)en für die Unterstützung der

- Programmierung im Großen.

In diesem Kapitel

- Ziele und Kennzeichen guter Modularisierung (Kap. 17.1)
- Haskells Modulkonzept (Kap. 17.2)
- Spezielle Anwendung: Abstrakte Datentypen (Kap. 17.3)

Kapitel 17.1

Ziele und Richtlinien guter Modularisierung

Modularisierung

Intuitiv:

- Zerlegung großer Programm(system)e in kleinere Einheiten, genannt Module.

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems.

Modularisierungsgewinne

Vorteile:

- Arbeitsphysiologisch: Unterstützung arbeitsteiliger Programmierung.
- Softwaretechnisch: Unterstützung der Wiederbenutzung von Programmen und Programmteilen.
- Implementierungstechnisch: Unterstützung getrennter Übersetzung (engl. separate compilation).

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Qualitätssteigerung (Verlässlichkeit) und Kostenreduktion.

Modularisierungsanforderungen

...zur Erreichung vorgenannter Ziele.

Unterstützung des Geheimnisprinzips durch Trennung von

- Schnittstelle (Import/Export)
  - Wie interagiert das Modul mit seiner Umgebung?
  - Welche Funktionalität stellt es zur Verfügung (Export)?
  - Welche Funktionalität benötigt es (Import)?
- Implementierung (Daten/Funktionen)
  - Wie sind die Datenstrukturen implementiert?
  - Wie ist die Funktionalität auf den Datenstrukturen realisiert?

Regeln "guter" Modularisierung

Lokale Sicht: Jedes Modul soll

- einen klar definierten, unabhängig von anderen Modulen verständlichen Zweck besitzen.
- nur einer Abstraktion entsprechen.
- einfach zu testen sein.

Globale Sicht: In modular entworfenen Programmen sollen

- Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung)
- Abhängigkeiten von anderen Programmen oder Hardware

...auf wenige Module beschränkt sein.

Modularisierungseigenschaften

...zentral:

- Intramodular: Kohäsion
  - beschäftigt sich mit Art und Typ der in einem Modul zusammengefassten Funktionen.
- Intermodular: Koppelung
  - beschäftigt sich mit dem Import-/Export- und Datenaustauschverhalten von Modulen.

Intramodular: Kohäsion

...anzustreben:

- Funktionale Kohäsion (Funktionen gleicher Funktionalität sind in einem Modul zusammengefasst, z.B. Sortierverfahren, Ein-/Ausgabefunktionen, etc.).
- Datenkohäsion (Auf gleichen Datenstrukturen arbeitende Funktionen sind in einem Modul zusammengefasst, z.B. Funktionen auf trigonometrischen Daten, auf Wasserstandsdaten, etc.).

...zu vermeiden:

- Logische Kohäsion (Funktionen vergleichbarer Funktionalität mit unterschiedlicher Implementierung sind in einem Modul zusammengefasst, z.B. verschiedene Benutzerschnittstellen eines Systems).
- Zufällige Kohäsion (Funktionen sind sachlich unbegründet in einem Modul zusammengefasst, zufällig eben).

Intermodular: Koppelung

...anzustreben:

- Schwache funktionale Koppelung, d.h. wenige, wohlbegründete funktionale Beziehungen und Abhängigkeiten zwischen verschiedenen Modulen.
- Feste Datenkoppelung, d.h. durch Wertübergabe (Funktionen unterschiedlicher Module kommunizieren nur durch explizite Übergabe von Werten, d.h. Ergebnisse einer Funktion werden Argument einer anderen Funktion.).

..zu vermeiden:

- Starke funktionale Koppelung.
- Lose Datenkoppelung, d.h. durch andere Mechanismen als Wertübergabe, z.B. Kommunikation über Dateien.

Bemerkung: Datenkoppelung durch Wertübergabe ist in funktionalen Sprachen *per se* als Grundform gegeben.

Ziel und Kennzeichen “guter” Modularisierung

Starke funktionale und Datenkohäsion

- enger inhaltlicher Zusammenhang der Definitionen eines Moduls.

Schwache funktionale und lose Datenkoppelung

- wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere keine direkten oder indirekten zirkulären Abhängigkeiten.

Für eine weitergehende und vertiefende Diskussion siehe:

- Manuel Chakravarty, Gabriele Keller. Einführung in die Programmierung mit Haskell, Pearson Studium, 2004, Kapitel 10.

Kapitel 17.2

Haskells Modulkonzept

Allgemeiner Aufbau eines Haskellmoduls

```
module M where                -- Moduldefinition

data D_1 ... = ...            -- Datentypdefinitionen
...
data D_n ... = ...

type T_1 ... = ...            -- Typsynonymdefinitionen
...
type T_m ... = ...

class C_1 ...                 -- Typklassendefinitionen
...
class C_k ...

f_1 :: ...                    -- Funktionsdefinitionen
f_1 ... = ...
...
f_p :: ...
f_p ... = ...
```

Haskells Modulkonzept

...unterstützt:

- Export
  - Selektiv/nicht selektiv
  - Händischer Reexport
  - Nicht unterstützt: Automatischer Reexport
- Import
  - Selektiv/nicht selektiv
  - Qualifiziert
  - Mit Umbenennung

von Datentypen, Typsynonymen, Typklassen und Funktionen.

Kapitel 17.2.1

Import

Import: Nicht selektiv

Allgemeines Muster:

```
module M1 where
...

module M2 where
import M1 -- Alle im Modul M1 (global sichtbaren) Be-
          -- zeichner, Definitionen werden importiert
          -- und können im Modul M2 benutzt werden.
```

Import: Selektiv

Allgemeines Muster, zwei Varianten:

```
module M1 where...

module M2 where
import M1 (D_1 (..), D_2, T_1, C_1 (..), C_2, f_5)
 -- Ausschließlich D_1 (einschließlich von M1 expor-
 -- tierter Konstruktoren), D_2 (ohne Konstruktoren),
 -- T_1, C_1 (..) (einschließlich von M1 exportierter
 -- Funktionen), C_2 (ohne Funktionen), f_5 werden aus
 -- M1 importiert und können in M2 benutzt werden,
 -- d.h., importiere nur, was explizit genannt ist!

module M3 where
import M1 hiding (D_1, T_2, f_1)
 -- Alle (sichtbaren) Bezeichner, Definitionen aus M1
 -- mit Ausnahme der explizit genannten werden impor-
 -- tiert und können in M3 benutzt werden, d.h., impor-
 -- tiere alles, was nicht explizit ausgeschlossen wird!
```

Anwendungsbeispiel

..."verstecken" der im Standard-Präludium vordefinierten Funktionen

- `reverse`, `tail` und `zip`

durch Einfügen von

- `import Prelude hiding (reverse,tail,zip)`

am Anfang des Haskell-Skripts im Anschluss an die (so vorhandene) Modul-Anweisung (siehe Kapitel 1.3.1).

Kapitel 17.2.2

Export

Export: Nicht selektiv

Allgemeines Muster:

```
module M1 where        -- Alle im Modul M1 eingeführten
data D_1 ... = ...     -- global sichtbaren Bezeichner,
...                    -- Definitionen werden exportiert
data D_n ... = ...     -- und können von anderen Modulen
type T_1 = ...         -- importiert werden.
...
type T_m = ...
class C_1 ...
...
class C_k ...
f_1 :: ...
f_1 ... = ...          -- Beachte:
...                    -- module M1 where...
f_p :: ...             -- ist bedeutungsgleich zu
f_p ... = ...          -- module M1 (module M1) where...
```

Export: Selektiv

Allgemeines Muster:

```
module M1 (D_1 (..), D_2, D_3 (Dc_1,...,Dc_k), C_1 (..),
           C_2, C_3 (cf_1,...,cf_l), T_1, f_2, f_5) where

data D_1 ... = ...  -- Nur die explizit genannten Bezeich-
...                 -- ner, Definitionen werden aus M1 ex-
data D_n ... = ...  -- portiert und können von anderen Mo-
                    -- dulen importiert werden. Dabei gilt:
type T_1 ... = ...  -- D_1 wird einschließlich seiner Kon-
...                 -- struktoren exportiert; D_2 ohne; D_3
type T_m ... = ...  -- mit den explizit genannten. Analog
class C_1 ...       -- für die Klassen C_i.
...
class C_k ...       -- Selektiver Export unterstützt
f_1 :: ...          -- das Geheimnisprinzip!
f_1 ... = ...
...
f_p :: ...
f_p ... = ...
```

Kapitel 17.2.3

Reexport

Reexport: Nicht automatisch, nur händisch

Veranschaulichung:

```
module M1 where...

module M2 where
import M1 -- Nicht selektiver Import aus M1, d.h. alle
...       -- in M1 (global sichtbaren) Bezeichner, De-
f_2M2     -- finitionen werden von M2 importiert und
...       -- können in M2 benutzt werden.

module M3 where
import M2 -- Nicht selektiver Import aus M2, d.h. alle
...       -- in M2 (global sichtbaren) Bezeichner, De-
          -- finitionen werden von M3 importiert und
          -- können in M3 benutzt werden, nicht jedoch
          -- die von M2 aus M1 importierten Namen,
          -- d.h. kein automatischer Reexport!
```

Reexport: Händisch

Abhilfe: Händischer Reexport, zwei Varianten:

```
module M2 (module M1,f_2M2) where -- Nicht selektiver Reex-
import M1                         -- port von M1 aus M2:
...                               -- M2 reexportiert jeden
f_2M2                             -- aus M1 importierten
                                  -- Namen, sowie das M2-
                                  -- lokale f_2M_2 aus M2.
...

module M2 (D_1 (..), D_2, D_3 (Dc_1,Dc_2), C_1 (..), C_2,
       C_3 (cf_1,cf_2,cf_3), f_1,f_2M2) where
import M1  -- Selektiver Reexport von M1 aus M2: M2 reex-
...        -- portiert von den aus M1 importierten Namen
f_2M2      -- ausschließlich D_1 (einschließlich Konstruk-
...        -- toren), D_2 (ohne Konstruktoren), D_3 (mit
           -- angegebenen Konstruktoren); analog für die
           -- Klassen C_1, C_2 und C_3, sowie f_1 und das
           -- M2-lokale f_2M_2.
```

Kapitel 17.2.4

Namenskonflikte, Umbenennungen, Konventionen

Namenskonflikte, Umbenennungen

Namenskonflikte

- können durch qualifizierten Import aufgelöst werden:

  `import qualified M1`

  Verwendung: `M1.f` zur Bezeichnung der aus `M1` importierten Funktion `f`; `f` zur Bezeichnung der im importierenden Modul lokal definierten Funktion `f`.

Umbenennen importierter Module und Bezeichner

- durch Einführen lokaler Namen im importierenden Modul
  - für Modulnamen:

    `import qualified M1` as MyLocalNameForM1

    ...`MyLocalNameForM1` wird im importierenden Modul anstelle von `M1` verwendet.
  - für ausgewählte Bezeichner:

```
import M1 (f1,f2)
  renaming (f1 to ggt, f2 to kgv)
```

Konventionen, gute Praxis

Konventionen

- Pro Datei ein Modul.
- Modul- und Dateiname stimmen überein (abgesehen von der Endung `.hs` bzw. `.lhs` im Dateinamen).
- Alle Deklarationen beginnen in derselben Spalte wie das Schlüsselwort `module`.

Gute Praxis

- Module unterstützen eine (!) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium).
- Module sind "kurz" ("so kurz wie möglich, so lang wie nötig"; ideal: zwei oder drei Bildschirmseiten).

Haskell-Programme

...sind Modulsysteme.

Soll ein Haskell-Programm übersetzt (statt interpretiert) werden, muss dessen Modulsystem ein Hauptmodul namens

- `Main`

mit einer Funktion namens

- `main :: IO` τ für einen Typ τ

enthalten, mit deren Auswertung die Ausführung des übersetzten Programms beginnt (wobei das Ergebnis vom Typ τ unbeachtet bleibt).

Bemerkung: Die `module`-Deklaration darf in einem Haskell-Skript fehlen; implizit wird in diesem Fall die `module`-Deklaration

```
module Main (main) where
```

ergänzt.

Kapitel 17.3

Spezielle Anwendung: Abstrakte Datentypen

Abstrakte Datentypen (ADTs)

...als Anwendung des Modulkonzepts in Haskell.

Mit ADTs verfolgtes Ziel:

- Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. information hiding).

Implementierungstechnischer Schlüssel:

- Haskells Modulkonzept, speziell selektiver Export, bei dem Konstruktoren algebraischer Datentypen verborgen bleiben.

Grundlegende Idee von ADT-Definitionen (1)

...implizite Festlegung eines Datentyps in zwei Teilen:

- A) Schnittstellenfestlegung: Angabe der auf den Werten des Datentyps zur Verfügung stehenden Operationen in Form ihrer syntaktischen Signaturen.
- B) Verhaltensfestlegung: Festlegung der Bedeutung der Operationen durch Angabe ihres Zusammenspiels in Form von Axiomen (oder sog. Gesetzen), die von einer Implementierung dieser Operationen einzuhalten sind.

Beachte: Die Darstellung der Werte des abstrakten Datentyps wird ausdrücklich nicht festgelegt; sie bleibt verborgen und deshalb für die Implementierung als Freiheitsgrad offen!

Grundlegende Idee von ADT-Definitionen (2)

Herausforderung:

- Die Gesetze so zu wählen, dass das Verhalten der Operationen präzise und eindeutig festgelegt ist; also so, dass weder eine Überspezifikation (keine widerspruchsfreie Implementierung möglich) noch eine Unterspezifikation (mehrere in sich widerspruchsfreie, aber sich widersprechende Implementierungen möglich) vorliegt.

Pragmatischer Gewinn:

- Die Trennung von Schnittstellen- und Verhaltensfestlegung erlaubt die Implementierung zu verstecken (Geheimnisprinzip!) und nach Zweckmäßigkeit und Anforderungen (z.B. Einfachheit, Performanz) auszuwählen und auszutauschen.

Beispiel: FIFO-Warteschlange als ADT

...in Pseudo-Code (kein Haskell):

```
A: Festlegung der Schnittstelle durch Signaturangabe:
    NEW:                         -> Queue
    ENQUEUE:   Queue × Item      -> Queue
    FRONT:     Queue             -> Item
    DEQUEUE:   Queue             -> Queue
    IS_EMPTY:  Queue             -> Boolean

B: Festlegung der einzuhaltenden Axiome/Gesetze:
 a) IS_EMPTY(NEW)              = true
 b) IS_EMPTY(ENQUEUE(q,i))     = false
 c) FRONT(NEW)                 = error
 d) FRONT(ENQUEUE(q,i))        = if IS_EMPTY(q) then i
                                 else FRONT(q)
 e) DEQUEUE(NEW)               = error
 f) DEQUEUE(ENQUEUE(q,i))      = if IS_EMPTY(q) then NEW
                                 else ENQUEUE(DEQUEUE(q),i)
```

FIFO-Warteschlange: Haskell-Realisierung (1)

```
-- A) Schnittstellenspezifikation:

module Queue
  (Queue,    -- Name des Datentyps (Geheimnisprinzip, kein
             -- Konstruktorexport!)
   new,      -- new :: Queue a
   enqueue,  -- enqueue :: Queue a -> a -> Queue a
   front,    -- front :: Queue a -> a
   dequeue,  -- dequeue :: Queue a -> Queue a
   is_empty, -- is_empty :: Queue a -> Bool
   -- a) is_empty(new) = True
   -- b) is_empty(enqueue(q,i)) = False
   -- c) front(new) = error "Nobody is waiting!"
   -- d) front(enqueue(q,i)) =
          if is_empty(q) then i else front(q)
   -- e) dequeue(new) = error "Nobody is waiting!"
   -- f) dequeue(enqueue(q,i)) =
          if is_empty(q) then new else enqueue(dequeue(q),i)
  ) where...
```

FIFO-Warteschlange: Haskell-Realisierung (2)

```
-- B1) Implementierung als algebraischer Datentyp:

data Queue a = Qu [a]

new :: Queue a
new = Qu []

enqueue :: Queue a -> a -> Queue a
enqueue (Qu xs) x = Qu (xs ++ [x])

front :: Queue a -> a
front q@(Qu xs)
 | not (is_empty q) = head xs
 | otherwise        = error "Schlange leer; niemand wartet!"

dequeue :: Queue a -> Queue a
dequeue q@(Qu xs)
 | not (is_empty q) = Qu (tail xs)
 | otherwise        = error "Schlange leer; niemand wartet!"

is_empty :: Queue a -> Bool
is_empty (Qu []) = True
is_empty _       = False
```

FIFO-Warteschlange: Haskell-Realisierung (3)

```
-- B2) Implementierung als neuer Typ:

newtype Queue a = Qu [a]

new :: Queue a
new = Qu []

enqueue :: Queue a -> a -> Queue a
enqueue (Qu xs) x = Qu (xs ++ [x])

front :: Queue a -> a
front q@(Qu xs)
 | not (is_empty q) = head xs
 | otherwise        = error "Schlange leer; niemand wartet!"

dequeue :: Queue a -> Queue a
dequeue q@(Qu xs)
 | not (is_empty q) = Qu (tail xs)
 | otherwise        = error "Schlange leer; niemand wartet!"

is_empty :: Queue a -> Bool
is_empty (Qu []) = True
is_empty _       = False
```

FIFO-Warteschlange: Haskell-Realisierung (4)

```
-- B3) Implementierung als Typsynonym gewöhnlicher Listen:

type Queue a = [a]

new :: Queue a
new = []

enqueue :: Queue a -> a -> Queue a
enqueue q x = q ++ [x]

front :: Queue a -> a
front q
 | not (is_empty q) = head q
 | otherwise        = error "Schlange leer; niemand wartet!"

dequeue :: Queue a -> Queue a
dequeue q
 | not (is_empty q) = tail q
 | otherwise        = error "Schlange leer; niemand wartet!"

is_empty :: Queue a -> Bool
is_empty q = (q == [])
```

Erinnerung: Das "als"-Muster

...als nützliche Musterspielart (siehe Kapitel 6.1.5):

```
front q@(Qu xs)   -- Arg. als q oder als (Qu xs).
dequeue q@(Qu xs) -- Arg. als q oder als Qu xs).
```

Das "als"-Muster (`q@(Qu xs)`) erlaubt mittels:

- `q`: Zugriff auf das Argument als Ganzes.
- `(Qu xs)`: Zugriff auf Teile des strukturierten Arguments.

Abstrakte vs. algebraische Datentypen

Abstrakte Datentypen

- werden durch ihr Verhalten spezifiziert, d.h. durch die auf ihren Werten definierten Funktionen/Operationen und deren Zusammenspiel.
- die Darstellung der Werte des Datentyps wird zum Definitionszeitpunkt nicht angegeben und bleibt offen.

Algebraische Datentypen

- werden durch die Angabe ihrer Elemente spezifiziert, aus denen sie bestehen.
- auf ihnen gegebene Funktionen/Operationen werden zum Definitionszeitpunkt nicht angegeben und bleiben offen.

Programmiertechnische Vorteile

...aus der Benutzung von ADTs:

- Geheimnisprinzip: Nur die Schnittstelle ist bekannt, die Implementierung bleibt verborgen.
  - Schutz der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff.

    Beispiel: Ein eigendefinierter Leerheitstest wie etwa

    ```
    emptyQ == Qu []
    ```

    führte in `Queue` importierenden Modulen zu einem Laufzeitfehler, da die Implementierung und somit der Konstruktor `Qu` dort nicht sichtbar sind.
  - Einfache Austauschbarkeit der zugrundeliegenden Implementierung.
  - Unterstützung arbeitsteiliger Programmierung.

Zur ADT-Realisierung in Haskell-Realisierung

...ADTs keine erstrangigen Sprachelemente (engl. first class citizens) in Haskell:

- Haskell bietet kein dezidiertes Sprachkonstrukt zur Spezifikation von ADTs, das eine externe Offenlegung von Signaturen und Gesetzen bei intern bleibender Implementierung erlaubte.
- ADTs mittels Modulen in Haskell zu spezifieren, ermöglicht die Implementierung intern und versteckt zu halten, jedoch können die Signaturen und Gesetze nur in Form von Kommentaren offengelegt werden.

Wegweisende Arbeiten

...zu abstrakten Datentypen:

- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Kapitel 17.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 8, Modularisierung und Schnittstellen)

- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 10, Modularisierung und Programmdekomposition)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 6, Modules)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (2)

- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (3)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 5, Writing a Library: Working with JSON Data – The Anatomy of a Haskell Module, Generating a Haskell Program and Importing Modules)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 14, Datenstrukturen und Modularisierung)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 15.1, Modules in Haskell; Kapitel 15.2, Modular design; Kapitel 16, Abstract data types)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (4)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 15.1, Modules in Haskell; Kapitel 15.2, Modular design; Kapitel 16, Abstract data types)

Kapitel 18

Programmierprinzipien

Programmierprinzipien

Funktionen höherer Ordnung (Kap. 18.1)

- ermöglichen algorithmisches Vorgehen zu verpacken.

  Illustrierendes Beispiel: Teile und Herrsche.

Verzögerte Auswertung (engl. lazy evaluation) (Kap. 18.2)

- ermöglichen semantische Modularisierungsprinzipien: Generator/Selektor-, Generator/Filter-, Generator/Transformator-Prinzip und Kombinationen davon.

  Illustrierendes Beispiel: Programmieren mit Strömen (unendliche Listen (engl. streams, lazy lists)).

Reflektives Programmieren (Kap. 18.3)

- Stetes Hinterfragen und Anpassen des eigenen Vorgehens.

Kapitel 18.1

Teile und Herrsche

Teile und Herrsche

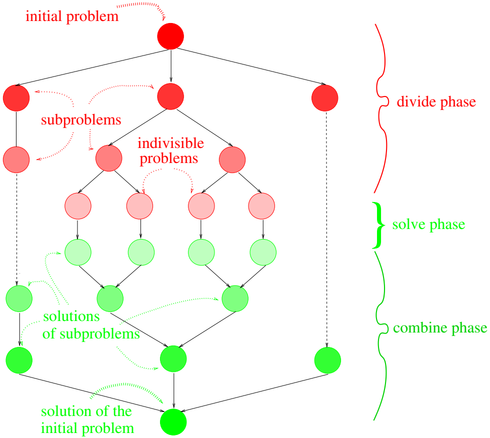
...zugrundeliegende algorithmische Idee des "Teile und Herrsche"-Prinzips:

- Ist ein Problem einfach genug, so löse es sofort.
- Anderenfalls zerlege das Problem in kleinere Teilprobleme und wende die Zerlegungsstrategie rekursiv an, bis alle Teilprobleme einfach genug sind zur sofortigen Lösung.
- Berechne die Lösung des ursprünglichen Problems aus den Lösungen der Teilprobleme.

...eine typische *top-down* Vorgehensweise!

Veranschaulichung

Die Phasenabfolge eines "Teile und Herrsche"-Algorithmus:



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, Seite 156.

Typische Anwendungsfelder

...und Anwendungen für Vorgehen mittels teile und herrsche:

- Sortierverfahren (Quicksort, Mergesort, etc.)
- Binomialkoeffizientenberechnung
- Numerische Analyseverfahren
- Kryptography
- Bildverarbeitung
- ...

Vorbereitung der funktionalen Umsetzung

Gegeben:

- Ein Problem mit Probleminstanzen eines generischen Typs, beschrieben durch die Typvariable `pb`.

Gesucht:

- Eine Lösung aus einer Menge von Lösungsinstanzen eines generischen Typs, beschrieben durch die Typvariable `lsg`.

(Algorithmisches) Ziel:

Eine Funktion höherer Ordnung (oder Funktional) `teile_und_herrsche`, die geeignet parametrisiert für

- Probleminstanzen vom Typ `pb` gemäß des "Teile und Herrsche"-Prinzips eine Lösungsinstanz vom Typ `lsg` berechnet.

Parameter

... des Funktionals `teile_und_herrsche`:

- `einfach_genug :: pb -> Bool`: ...liefert `True`, falls die Probleminstanz einfach genug ist, um sofort gelöst werden zu können.
- `loese :: pb -> lsg`: ...liefert die Lösungsinstanz einer unmittelbar lösbaren Probleminstanz.
- `teile :: pb -> [pb]`: ...teilt eine nicht unmittelbar lösbare Probleminstanz in eine Liste von Teilprobleminstanzen auf.
- `herrsche :: pb -> [lsg] -> lsg`: ...liefert angewendet auf eine Ausgangsprobleminstanz und eine Liste von Lösungen von Teilprobleminstanzen die Lösung der Ausgangsprobleminstanz.

Nützliche Typsynonyme

...auftretender Funktionstypen:

```
type Einfach_genug pb  = pb -> Bool
type Loese pb lsg      = pb -> lsg
type Teile pb          = pb -> [pb]
type Herrsche pb lsg   = pb -> [lsg] -> lsg
```

Das Funktional teile_und_herrsche

```
teile_und_herrsche :: (Einfach_genug pb) -> (Loese pb lsg)
     -> (Teile pb) -> (Herrsche pb lsg) -> pb -> lsg
teile_und_herrsche einfach_genug loese teile herrsche
                   pb_instanz
 = tuh pb_instanz
   where
    tuh p
     | einfach_genug p = loese p
     | otherwise       = herrsche p (map tuh (teile p))
```

Löse rekursiv alle durch die Teilung entstehenden Probleme.

Teile und Herrsche am Beispiel von Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort liste
 = teile_und_herrsche einfach_genug loese teile
                      herrsche liste
 where
  einfach_genug ls          = length ls <= 1
  loese                     = id
  teile (l:ls)              = [[ x | x <- ls, x <= l],
                                [ x | x <- ls, x > l]]
  herrsche (l:_) [ls1,ls2] = ls1 ++ [l] ++ ls2
```

Warnung

...nicht jedes Problem, das dem "teile und herrsche"-Vorgehen in natürlicher Weise zugänglich ist, ist auch (in naiver Weise) dafür geeignet.

Betrachte dazu folgendes Beispiel:

```
fib :: Integer -> Integer
fib n = teile_und_herrsche einfach_genug loese
                           teile herrsche n
   where
    einfach_genug n     = (n == 0) || (n == 1)
    loese               = id
    teile n             = [n-2,n-1]
    herrsche _ [m1,m2]  = m1 + m2
```

...besitzt exponentielles Laufzeitverhalten!

Algorithmenmuster

Die Idee, ein generelles algorithmisches Vorgehen wie "teile und herrsche" durch eine geeignete Funktion höherer Ordnung wiederverwendbar zu machen, lässt sich auch für andere algorithmische Verfahrensweisen umsetzen, darunter

- Rücksetzsuche (engl. Backtracking Search)
- Prioritätsgesteuerte Suche
- Lokale Suche (engl. Greedy Search)
- Dynamische Programmierung

Wir sprechen hier auch von Algorithmenmustern (mehr dazu in der LVA 185.A05 "Fortgeschrittene funktionale Programmierung").

Kapitel 18.2

Stromprogrammierung

Ströme

...programmiersprachlicher Jargon für

- unendliche Listen (engl. streams, lazy lists).

Ströme ermöglichen im Zusammenspiel mit verzögerter Auswertung

- neue, semantikbasierte Modularisierungen
  - Generator/Selektor-Prinzip
  - Generator/Filter-Prinzip
  - Generator/Transformator-Prinzip

mit denen sich viele Probleme elegant, knapp und effizient lösen lassen.

Ströme am Bsp. des Siebs des Eratosthenes (1)

...zur Berechnung des Stroms der Primzahlen:

1. Schreibe alle natürlichen Zahlen ab 2 hintereinander auf.
2. Die kleinste nicht gestrichene Zahl in dieser Folge ist eine Primzahl. Streiche alle Vielfachen dieser Zahl.
3. Wiederhole Schritt 2 mit der kleinsten jeweils noch nicht gestrichenen Zahl.

```
Nach Schritt 1:
 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17...
Nach Schritt 2 für "2":
 2  3     5     7     9     11    13    15    17...
Nach Schritt 2 für "3":
 2  3     5     7           11    13          17...
usw.
```

Ströme am Bsp. des Siebs des Eratosthenes (2)

```
sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, mod y x > 0]

primes :: [Integer]
primes = sieve [2..]
```

Die (0-stellige) Funktion

- primes liefert den Strom der (unendlich vielen) Primzahlen.
- Aufruf:

  ```
  primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,...
  ```

Ströme am Bsp. des Siebs des Eratosthenes (3)

Veranschaulichung der Stromberechnung durch händische Auswertung:

```
    primes
->> sieve [2..]
->> 2 : sieve [y | y <- [3..], mod y 2 > 0]
->> 2 : sieve (3 : [y | y <- [4..], mod y 2 > 0]
->> 2 : 3 : sieve [z | z <- [ y | y <- [4..],
                               mod y 2 > 0],
                               mod z 3 > 0]
->> ...
->> 2 : 3 : sieve [z | z <- [5, 7, 9..],
                      mod z 3 > 0]
->> ...
->> 2 : 3 : sieve [5, 7, 11,...
->> ...
```

Semantische Modularisierungsprinzipien

...aus dem Stromkonzept erwachsen neue, semantische Modularisierungsprinzipien.

Insbesondere:

- Generator/Selektor- (G/S-) Prinzip
- Generator/Filter- (G/F-) Prinzip
- Generator/Transformator- (G/T-) Prinzip

sowie Kombinationen davon wie das G/T/S- und G/T/F-Prinzip und weitere.

G/S-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ```
  genPrimes :: [Integer]
  genPrimes = primes
  ```

Viele Selektoren (S):

- Nimm die ersten n Elemente einer Liste:
  ```
  take :: Int -> [a] -> [a]
  take n lst = ...
  ```
- Nimm das $(n-1)$-te Element einer Liste:
  ```
  !! :: [a] -> Int -> a
  (!!) lst n = ...
  ```
- Nimm alle ab dem $(n+1)$-ten Element einer Liste:
  ```
  drop :: Int -> [a] -> [a]
  drop n lst = ...
  ```
- ...
  ```

G/S-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/S-Module zum Gesamtprogramm:

- Anwendung des G/S-Prinzips:

  Die ersten 5 Primzahlen:

  `take 5 genPrimes ->> [2,3,5,7,11]`

- Anwendung des G/S-Prinzips:

  Die 5-te Primzahl:

  `(!!) 4 genPrimes ->> genPrimes!!4 ->> 11`

- Anwendung des G/S-Prinzips:

  Die 6-te bis 10-te Primzahl:

  `take 5 (drop 5 genPrimes) ->> [13,17,19,23,29]`

G/F-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ```
  genPrimes :: [Integer]
  genPrimes = primes
  ```

Viele Filter (F):

- Alle Listenelemente größer als 1000:
  ```
  filter (>1000) :: [Integer] -> [Integer]
  filter (>1000) lst = ...
  ```
- Ist Zahl mit genau drei Einsen in der Dezimaldarstellung:
  ```
  hatDreiEinsen :: Integer -> Bool
  hatDreiEinsen n = ...
  ```
- Ist Zahl mit Palindromdezimaldarstellung:
  ```
  istPalindrom :: Integer -> Bool
  istPalindrom n = ...
  ```
- ...
  ```

G/F-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/F-Module zum Gesamtprogramm:

- Anwendung des G/F-Prinzips:

  Alle Primzahlen größer als 1000:

  ```
  filter (>1000) genPrimes
    ->> [1009,1013,1019,1021,1031,1033,1039,...
  ```

- Anwendung des G/F-Prinzips:

  Alle Primzahlen mit genau drei Einsen in der Dezimaldarstellung:

  ```
  [ n | n <- genPrimes, hatDreiEinsen n]
    ->> [1117,1151,1171,1181,1511,1811,2111,...
  ```

- Anwendung des G/F-Prinzips:

  Alle Primzahlen mit Palindromdezimaldarstellung:

  ```
  [ n | n <- genPrimes, istPalindrom n]
    ->> [2,3,5,7,11,101,131,151,181,191,313,...
  ```

G/T-Prinzip am Bsp. des Primzahlstroms (1)

Ein Generator (G):

- ```
  genPrimes :: [Integer]
  genPrimes = primes
  ```

Viele Transformatoren (T):

- Quadrieren (für den Strom der Quadratprimzahlen):
  ```
  square :: Integer -> Integer
  square n = ...
  ```
- Dekrementieren (für den Strom der Primzahlvorgänger):
  ```
  decrement :: Integer -> Integer
  decrement n = n-1
  ```
- Summieren (für den Strom der partiellen Primzahlsummen (den Strom d. Summen d. Primzahlen von 2 bis *n*)):
  ```
  sum :: [Integer] -> Integer
  sum lst = ...
  ```
- ...
  ```

G/T-Prinzip am Bsp. des Primzahlstroms (2)

Zusammenfügen der G/T-Module zum Gesamtprogramm:

- Anwendung des G/T-Prinzips:

  Der Strom der Quadratprimzahlen:

  ```
  [ square n | n <- genPrimes ]
     ->> [4,9,25,49,121,169,289,361,529,841,...
  ```

- Anwendung des G/T-Prinzips:

  Der Strom der Primzahlvorgänger:

  ```
  [ decrement n | n <- genPrimes ]
     ->> [1,2,4,6,10,12,16,18,22,28,...
  ```

- Anwendung des G/T-Prinzips:

  Der Strom der partiellen Primzahlsummen:

  ```
  [ sum [2..n] | n <- genPrimes ]
     ->> [2,5,14,27,65,90,152,189,275,434,...
  ```

Bemerkungen

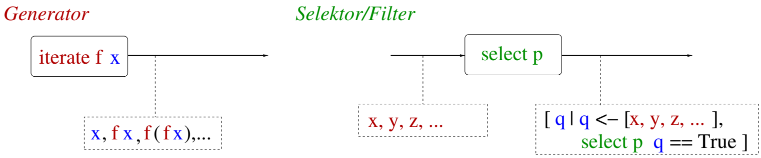
Auf Terminierung

- ist bei Anwendung der Prinzipiens stets besonders zu achten. So terminiert der Aufruf
  `filter (<10) genPrimes ->> [2,3,5,7,`
  nicht; der Aufruf
  `takeWhile (<10) genPrimes ->> [2,3,5,7]`
  hingegen schon.

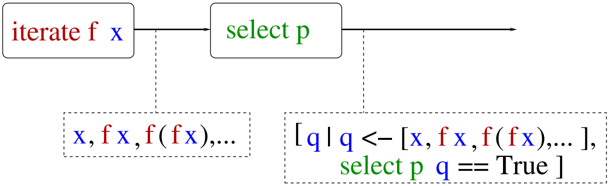
Nicht nur Generatoren

- lassen sich mit verschiedenen Selektoren, Filtern, Transformatoren verknüpfen wie in den Beispielen demonstriert, auch umgekehrt lassen sich Selektoren, Filter, Transformatoren mit verschiedenen Generatoren verknüpfen.

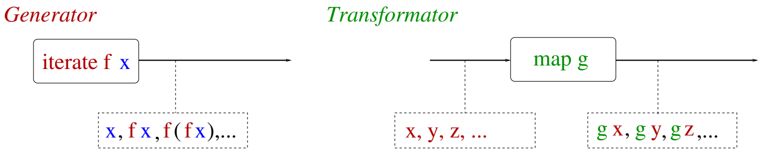
Das G/S- und G/F-Prinzip auf einen Blick



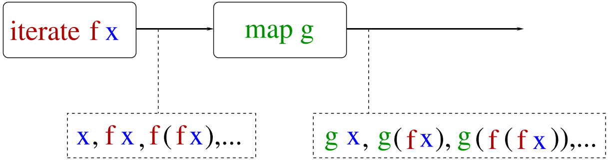
Verknüpfen von Generator und Selektor/Filter



Das G/T-Prinzip auf einen Blick



Verknüpfen von Generator und Transformator



Typische Anwendungen

...des G/S-, G/F- und G/T-Prinzips:

- Rucksackprobleme
- Pascalsches Dreieck
- Goldenes Verhältnis
- Fibonacci-Zahlen
- Potenzreihen
- ...

Fibonacci-Zahlen als Summe von Strömen (1)

Zwei Generatoren G1 und G2:

0	1	1	2	3	5	8	13...	G1: Strom der F.-Zahlen
1	1	2	3	5	8	13	21...	G2: Rest d. Stroms d. Fib.-Z.
+	+	+	+	+	+	+	+ + +	Summiere G1 und G2! + +
1	2	3	5	8	13	21	34...	Rest des Restes des Stroms der Fibonacci-Zahlen

Berechnung der Fibonacci-Zahlen als Summe von G1 und G2:

```
fibs :: [Integer] -- Generator der Fibonacci-Zahlen
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

$$\underbrace{0 : 1 : \underbrace{\texttt{zipWith (+)}\ \underbrace{\texttt{fibs}}_{\texttt{G1}}\ \underbrace{\texttt{(tail fibs)}}_{\texttt{G2}}}_{\texttt{Rest d. Restes d. Stroms d. Fib.-Z.}}}_{\texttt{Strom der Fibonacci-Zahlen}}$$

...sich wie Münchhausen "am eigenen Schopfe aus dem Sumpf ziehen"!

Fibonacci-Zahlen als Summe von Strömen (2)

Generator fibs:

```
fibs ->> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...
```

Generator/Filter-, Selektorkombinationen mit fibs:

```
filter even fibs ->> [0,2,8,34,144,...
fibs!!5 ->> 3
take 10 fibs ->> [0,1,1,2,3,5,8,13,21,34]
```

wobei

```
take :: Int -> [a] -> [a]
take 0 _              = []
take _ []             = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

Zusammenfassung

Verzögerte Auswertung (engl. lazy evaluation) erlaubt es

- die Kontrolle der Auswertungsreihenfolge von Daten

zu trennen und ermöglicht dadurch die elegante Behandlung

- unendlicher Datenwerte (genauer: nicht a priori in der Größe beschränkter Datenwerte), insbesondere
  - unendlicher Listen, sog. Ströme (engl. streams, lazy lists)

Dies führt zu semantikbasierten, von der Programmlogik her begründeten neuen Modularisierungsprinzipien:

- Generator/Selektor-Prinzip
- Generator/Filter-Prinzip
- Generator/Transformator-Prinzip

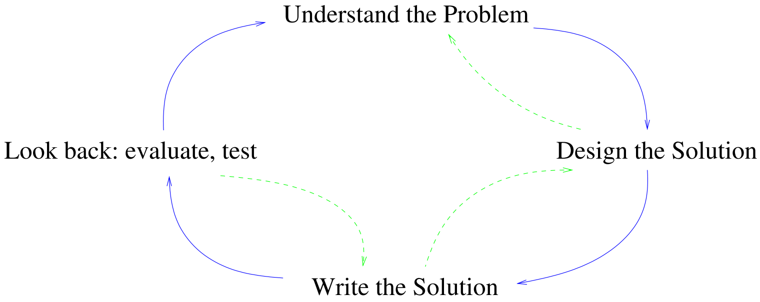
sowie von Kombinationen dieser Prinzipien.

Kapitel 18.3

Reflektives Programmieren

Reflektives Programmieren

...der Programm-Entwicklungszyklus nach Simon Thompson, Haskell: The Craft of Fuctional Programming, 2. Auflage, 1999, Kap. 11 “Reflective Programming”:



...in jeder der 4 Phasen ist es nützlich, (sich) Fragen zu stellen, zu beantworten und den Lösungsweg ggf. anzupassen.

Phase 1: Typische Fragen

Verstehen des Problems:

- Welches sind die Ein- und Ausgaben des Problems?
- Welche Randbedingungen sind einzuhalten?
- Ist das Problem über- oder unterspezifiziert?
- Ist das Problem entscheidbar und damit grundsätzlich lösbar? In welche Komplexitätsklasse fällt es?
- Ist das Problem aufgrund seiner Struktur in Teilprobleme zerlegbar?
- ...

Phase 2: Typische Fragen

Entwerfen einer Lösung:

- Ist das Problem verwandt zu (mir) bekannten anderen, möglicherweise einfacheren Problemen?
- Wenn ja, lassen sich deren Lösungsideen anpassen und anwenden? Ebenso deren Implementierungen, vorhandene Bibliotheken?
- Lässt sich das Problem verallgemeinern und so möglicherweise einfacher lösen?
- Ist das Problem mit den vorhandenen Ressourcen, einem gegebenen Budget lösbar?
- Ist die Lösung änderungs-, erweiterungs- und wiederbenutzungsfreundlich?
- ...

Phase 3: Typische Fragen

Ausformulieren und codieren der Lösung:

- Gibt es passende Bibliotheken, speziell geeignete polymorphe Funktionen höherer Ordnung für die Lösung von Teilproblemen?
- Können vorhandene Bibliotheksfunktionen (zumindest) als Vorbild dienen, um entsprechende Funktionen für eigene Datentypen zu definieren?
- Kann funktionale Abstraktion (auch höherer Stufe) zur Verallgemeinerung der Lösung angewendet werden?
- Welche Hilfsfunktionen, Datenstrukturen könnten nützlich sein?
- Welche Möglichkeiten der Sprache können für die Codierung vorteilhaft ausgenutzt werden und wie?
- ...

Phase 4: Typische Fragen

Eevaluieren, testen, Blick zurück:

- Lässt sich die Lösung testen, ihre Korrektheit auch formal beweisen?
- Worin sind möglicherweise gefundene Fehler begründet? Flüchtigkeitsfehler, Programmierfehler, falsches oder unvollständiges Problemverständnis, falsches Semantikverständnis der verwendeten Programmiersprache? Andere Gründe?
- Sollte das Problem noch einmal gelöst werden müssen; sollte die Lösung und ihre Implementierung genauso gemacht werden? Was sollte beibehalten oder geändert wer- den und warum?
- Erfüllt das Programm auch nichtfunktionale Eigenschaften gut wie Performanz, Speicherverbrauch, Skalierbarkeit, Verständlichkeit, Modifizier- und Erweiterbarkeit?
- ...

Kapitel 18.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 18 (1)

- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 9, Infinite Lists)
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 9, Infinite lists)
- Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.4, Divide and conquer; Kapitel 7, Infinite Lists)
- Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.2, Infinite Objects; Kapitel 7.3, Streams)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 18 (2)

- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 2.2, Unendliche Datenstrukturen)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 14, Programming with Streams)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 15.6, Modular programming)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 18 (3)

- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 20.2, Sortieren von Listen)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 2, Faulheit währt unendlich)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 8.1, Divide-and-conquer)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 18 (4)

- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 11, Program development; Kapitel 17, Lazy programming)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 12, Developing higher-order programs; Kapitel 17, Lazy programming)

Teil VII

Abschluss und Ausblick

Kapitel 19

Abschluss, Ausblick

Kapitel 19.1

Abschluss

Funktionale, imperative Programmierung (1)

Eigenschaften und Charakteristika im Vergleich.

- Funktional:
  - Programm ist Ein-/Ausgaberelation.
  - Programme sind zustandsfrei und 'zeitlos'.
  - Programmformulierung auf abstraktem, mathematisch geprägten Niveau, ohne eine Maschine im Blick.
- Imperativ:
  - Programm ist Arbeitsanweisung für eine Maschine.
  - Programme sind zustands- und 'zeitbehaftet'.
  - Programmformulierung mit Blick auf eine Maschine, ein Maschinenmodell (von Neumann).

Funktionale, imperative Programmierung (2)

- Funktional:
  - Die Auswertungsreihenfolge von Ausdrücken liegt (bis auf Datenabhängigkeiten) nicht fest.
  - Namen werden durch Wertvereinbarungen genau einmal für immer an einen Wert gebunden.
  - Schachtelung (rekursiver) Funktionsaufrufe erlaubt neue Werte mit neuen Namen zu verbinden.
- Imperativ:
  - Die Ausführungsreihenfolge von Anweisungen liegt fest; Freiheiten bestehen bei der Auswertungsreihenfolge von Ausdrücken (wie funktional).
  - Namen werden in der zeitlichen Abfolge durch Zuweisungen temporär mit Werten belegt.
  - Namen können durch wiederholte Zuweisungen beliebig oft mit neuen Werten belegt werden (in rekursiven Aufrufen, repetitiven Anweisungen wie *while*, *repeat*, *for*).

Möglichkeiten fkt. Programmierung

"Die Fülle an Möglichkeiten (in funktionalen Programmiersprachen) erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien."

Peter Pepper, *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003.

Für

- Funktionen
  - (Fkt.-) Applikation, Fallunterscheidung, Rekursion.
- Datenstrukturen
  - Aufzählung, Produkt- und Summenbildung, Rekursion.

Mächtigkeit fkt. Programmierung

...zusammen mit den durchgängigen Konzepten von

- Funktionen als erstrangige Sprachelemente (engl. first class citizens)
  - Funktionen höherer Ordnung
- Polymorphie auf
  - Funktionen
  - Datentypen

...führt dies zur Mächtigkeit und Eleganz funktionaler Programmierung, zusammengefasst im Slogan:

Functional Programming is Fun!

Zur (rethorischen) Eingangsfrage

> *"Can programming be liberated from the von Neumann style?"*
>
> John W. Backus, 1978

Ja (im Detail kann diskutiert werden, siehe Ein-/Ausgabe).

Erfolgreiche Einsatzfelder fkt. Programmierung

- Theorembeweiser HOL und Isabelle in ML.
- Modellprüfer (z.B. Edinburgh Concurrency Workbench).
- Mobility Server von Ericson in Erlang.
- Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML.
- Compiler in compilierter Sprache geschrieben.
- CPL/Kleisli (komplexe Datenbankabfragen) in ML.
- Natural Expert (Datenbankabfragen Haskell-ähnlich).
- Ensemble zur Spezifikation effizienter Protokolle (ML).
- Expertensysteme (insbesondere Lisp-basiert).
- ...
- `http://homepages.inf.ed.ac.uk/wadler/realworld`
- `www.haskell.org/haskellwiki/Haskell_in_industry`

Rückblick auf die Vorbesprechung

...warum die nächste Sprache funktional sein sollte:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4): 86-90, 2009.

  ...adopting a functional programming style could make your programs more robust, more compact, and more easily parallelizable.

- Konstantin Läufer, Geoge K. Thiruvathukal. The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering 11(5): 68-75, 2009.

  ...this second installment picks up where Konrad Hinsen's article "The Promises of Functional Programming" [...] left off, covering static type inference and lazy evaluation in functional programming languages.

Kapitel 19.2

Ausblick

“Alles, was man wissen muss,
um selber weiter zu lernen”.

Frei nach (aber im Sinne von)
Dietrich Schwanitz

Fort- und weiterführendes zu funktionaler Programmierung in TUW-Lehrveranstaltungen, insbesondere:

- LVA 185.A05 Fortgeschrittene funktionale Programmierung, VU 2.0, ECTS 3.0.
- Möglicherweise: LVA 127.008 Haskell-Praxis: Programmieren mit der funktionalen Programmiersprache Haskell VU 2.0, ECTS 3.0, Prof. em. Andreas Frank, Institut für Geoinformation und Kartographie.

LVA 185.A05 Fortg. fkt. Programmierung

Vorlesungsinhalte:

- Programmieren mit
  - Strömen, Funktoren, Monaden, Kombinatorbibliotheken.
  - Funktionalen Feldern, abstrakten Datentypen.
- Anwendungen
  - Funktionale reaktive Programmierung, logische Programmierung funktional, Parsing, funktionale Perlen, Algorithmenmuster.
- Qualitätssicherung
  - Programmverifikation und -validation, gleichungsbasiertes Schließen und Beweisen, automatisches Testen.
- ...

LVA 127.008 Haskell-Praxis [...]

Vorlesungsinhalte:

- Analyse und Verbesserung von gegebenem Code.
- Weiterentwicklung der Open-Source-Entwicklungsumgebung LEKSAH für Haskell, insbesondere der graphischen Benutzerschnittstelle (GUI).
- Gestaltung graphischer Benutzerschnittstellen (GUIs) mit Glade und Gtk+.
- ...

Always look on the bright side of life

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.(*)

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

(*) Zitat aus: Introducing a course on calculi. Ankündigung einer Lehrveranstaltung an der University of Texas at Austin, 1995.

Kapitel 19.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 19 (1)

- Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), 2003. `research.microsoft.com/users/simonpj/papers/haskell-retrospective/`
- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007. (ACM Digital Library `www.acm.org/dl`)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 19 (2)

- Andrew Appel. *A Critique of Standard ML*. Journal of Functional Programming 3(4):391-430, 1993.
- Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 5, Alternative functional styles)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 1.3, Features of Haskell; Kapitel 1.4, Historical background)
- Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009. (Kapitel 3, Programmiersprachen)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 19 (3)

- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 1, Introduction; Kapitel 9, Functional programming in Standard ML; Kapitel 10, Functional programming and LISP)
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 23, Compiler and Interpreter für Opal, ML, Haskell, Gofer)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 1.2, Functional Languages)
- Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 19 (4)

- Dietrich Schwanitz. *Bildung: Alles, was man wissen muss*. Eichborn Verlag, 1999.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Anhang A, Functional, imperative and OO programming)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Anhang A, Functional, imperative and OO programming)
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

Literaturverzeichnis

Literaturhinweise und Leseempfehlungen

...zum vertiefenden und weiterführenden Selbststudium.

- I Lehrbücher
- II Tutorien, Manuale
- III Grundlegende, wegweisende Artikel
- IV Weitere Arbeiten
- V Zum Haskell-Sprachstandard
- VI Die Haskell-Geschichte

Lehrbücher (1)

- Henri E. Baal, Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.
- Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North-Holland, 1984.
- Henrik P. Barendregt, Wil Dekkers, Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2012.
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.
- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998.

Inhalt
Kap. 1
Kap. 2
Kap. 3
Kap. 4
Kap. 5
Kap. 6
Kap. 7
Kap. 8
Kap. 9
Kap. 10
Kap. 11
Kap. 12
Kap. 13
Kap. 14
Kap. 15
Kap. 16
Kap. 17
Kap. 18

I Lehrbücher (2)

- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011.
- Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.
- Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012.

I Lehrbücher (3)

- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004.
- Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011.
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.
- Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
- Matthias Felleisen, Rober B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

I Lehrbücher (4)

- Hugh Glaser, Chris Hankin, David Till. *Principles of Functional Programming*. Prentice Hall, 1984.
- Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004.
- Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall, 1980.
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016.
- Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009.

I Lehrbücher (5)

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. `learnyouahaskell.com`
- Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
- Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. `book.realworldhaskell.org`
- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003.

I Lehrbücher (6)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-V., 2006.
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.
- Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
- Peter Rechenberg, Gustav Pomberger (Hrsg.). *Informatik-Handbuch*. Carl Hanser Verlag, 4. Auflage, 2006.
- Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.

I Lehrbücher (7)

- Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999.
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.
- Allen B. Tucker (Editor-in-Chief). *Computer Science. Handbook*. Chapman & Hall/CRC, 2004.
- Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.
- Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010.

II Tutorien, Manuale (1)

- H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. `citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf`
- Hal Daumé III. *Yet Another Haskell Tutorial*. `wikibooks.org`-Ausgabe, 2007. `https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial`
- Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. `tryhaskell.org`.
- Paul Hudak, Joseph Fasel, John Peterson. *A Gentle Introduction to Haskell*. Technischer Bericht, Yale University, 1996. `https://www.haskell/org/tutorial`

II Tutorien, Manuale (2)

- Hugs-Benutzerhandbuch. *The Hugs98 User Manual.* https://www.haskell.org/hugs/pages/hugsman/index.html
- GHCi-Benutzerhandbuch. *Glasgow Haskell Compiler User's Guide.* http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html
- Haskells Standard-Präludium. https://www.haskell.org/onlinereport/standard-prelude.html

III Grundlegende, wegweisende Artikel (1)

- John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
- Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
- Robert W. Floyd. *The Paradigms of Programming*. Turing Award Lecture, Communications of the ACM 22(8):455-460, 1979.
- John Hughes. *Why Functional Programming Matters*. The Computer Journal 32(2):98-107, 1989.

III Grundlegende, wegweisende Artikel (2)

- Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
- Christopher Strachey. *Fundamental Concepts in Programming Languages*. Higher-Order and Symbolic Computation 13:11-49, 2000, Kluwer Academic Publishers (revised version of a report of the NATO Summer School in Programming, Copenhagen, Denmark, 1967.)
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

IV Weitere Arbeiten (1)

- Andrew Appel. *A Critique of Standard ML*. Journal of Functional Programming 3(4):391-430, 1993.
- Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 233-246, 1995.
- Hendrik Pieter Barendregt, Erik Barendsen. *Introduction to the Lambda Calculus*. Revised Edn., Technical Report, University of Nijmegen, March 2000. `ftp://ftp.cs.kun.nl/pub/CompMath.Found/lambda.pdf`
- Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.

IV Weitere Arbeiten (2)

- Iavor S. Dachki, Thomas Hallgren, Mark P. Jones, Rebekah Leslie, Andrew Tolmach. *Writing System Software in a Functional Language: An Experience Report*. In Proceedings of the 4th International Workshop on Programming Languages and Operating Systems (PLOS 2007), Article No. 1, 5 pages, 2007.
- Luís Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.

IV Weitere Arbeiten (3)

- Noah M. Daniels, Andrew Gallant, Norman Ramsey. *Experience Report: Haskell in Computational Biology*. In Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012), 227-234, 2012.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC

IV Weitere Arbeiten (4)

- Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 11 pages, 2013. `www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf`
- Robert M. French. *Moving Beyond the Turing Test*. Communications of the ACM 55(12):74-77, 2012.
- Hugh Glaser, Pieter H. Hartel, Paul W. Garrat. *Programming by Numbers: A Programming Method for Novices*. The Computer Journal 43(4):252-265, 2000.
- Benjamin Goldberg. *Functional Programming Languages*. ACM Computing Surveys 28(1):249-251, 1996.

IV Weitere Arbeiten (5)

- Andrew J. Gordon. *Functional Programming and Input/Output*. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
- John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
- John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

IV Weitere Arbeiten (6)

- Bastiaan Heeren, Daan Leijen, Arjan van IJzendoorn. *Helium, for Learning Haskell*. In Proceedings of the ACM SIGPLAN 2003 Haskell Workshop (Haskell 2003), 62-71, 2003.
- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
- C.A.R. Hoare. *Algorithm 64: Quicksort*. Communications of the ACM 4(7):321, 1961.
- C.A.R. Hoare. *Quicksort*. The Computer Journal 5(1):10-15, 1962.
- Paul Hudak, Joseph H. Fasel. *A Gentle Introduction to Haskell*. ACM SIGPLAN Notices 27(5):1-52, 1992.
- Arjan van IJzendoorn, Daan Leijen, Bastiaan Heeren. *The Helium Compiler*. `www.cs.uu.nl/helium`.

IV Weitere Arbeiten (7)

- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.
- Donald Knuth. *Literate Programming*. The Computer Journal 27(2):97-111, 1984.
- Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.
- John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Electronic Notes in Theoretical Computer Science 1:370-392, 1995.

IV Weitere Arbeiten (8)

- John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. Journal of Functional Programming 8(3):275-317, 1998.
- John Maraist, Martin Odersky, David N. Turner, Philip Wadler. *Call-by-name, Call-by-value, call-by-need, and the Linear Lambda Calculus*. Theoretical Computer Science 228(1-2):175-210, 1999.
- Donald Michie. *'Memo' Functions and Machine Learning*. Nature 218:19-22, 1968.
- Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17:248-375, 1978.

IV Weitere Arbeiten (9)

- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
- John C. Mitchell. *Type Systems for Programming Languages*. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Jan van Leeuwen (Hrsg.). Elsevier Science Publishers, 367-458, 1990.
- William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.
- Gordon Plotkin. *Call-by-name, Call-by-value, and the λ-Calculus*. Theoretical Computer Science 1:125-159, 1975.

IV Weitere Arbeiten (10)

- Norman Ramsey. *On Teaching How to Design Programs*. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014), 153-166, 2014.
- J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1):23-42, 1965.
- Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*, Susan Eisenbach (Hrsg.), Ellis Horwood, 9-20, 1987.
- Uwe Schöning, Wolfgang Thomas. *Turings Arbeiten über Berechenbarkeit – eine Einführung und Lesehilfe*. Informatik Spektrum 35(4):253-260, 2012.

IV Weitere Arbeiten (11)

- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- Simon Thompson. *Where Do I Begin? A Problem Solving Approach in Teaching Functional Programming*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-Verlag, LNCS 1292, 323-334, 1997.
- Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2):25-30, 1998.

IV Weitere Arbeiten (12)

- Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8):23-27, 1998.
- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.

V Zum Haskell-Sprachstandard

- Paul Hudak, Simon Peyton Jones, Philip Wadler (Hrsg.). *Report on the Programming Language Haskell: Version 1.1*. Technical Report, Yale University and Glasgow University, August 1991.
- Paul Hudak, Simon Peyton Jones, Philip Wadler (Hrsg.) *Report on the Programming Language Haskell: A Non-strict Purely Funcional Language (Version 1.2)*. ACM SIGPLAN Notices, 27(5):1-164, 1992.
- Simon Marlow (Hrsg.). *Haskell 2010 Language Report*, 2010.
  `www.haskell.org/definition/haskell2010.pdf`
- Simon Peyton Jones (Hrsg.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. `www.haskell.org/definitions`.

VI Die Haskell-Geschichte

- Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), 2003. research.microsoft.com/users/simonpj/papers/haskell-retrospective/

- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007. (ACM Digital Library www.acm.org/dl)

Anhänge

A

Formale Rechenmodelle

A.1

Turing-Maschinen

Definition A.1.1 (Turing-Maschine)

- Eine Turing-Maschine TM ist ein "schwarzer" Kasten, der über einen Lese-/Schreibkopf mit einem (unendlichen) Rechenband verbunden ist.
- Das Rechenband ist in einzelne Felder eingeteilt, von denen zu jeder Zeit genau eines vom Lese-/Schreibkopf beobachtet wird.
- Es gibt eine Möglichkeit, TM einzuschalten; das Abschalten erfolgt selbsttätig.

Arbeitsweise einer Turing-Maschine

Eine Turing-Maschine TM kann folgende Aktionen ausführen:

- TM kann Zeichen $a_1, a_2, \ldots, a_n$ eines Zeichenvorrats $\mathcal{A}$ sowie das Sonderzeichen *blank* $\notin \mathcal{A}$ auf Felder des Rechenbandes drucken; *blank* steht dabei für das Leerzeichen.
- Dabei wird angenommen, dass zu jedem Zeitpunkt auf jedem Feld des Bandes etwas steht und dass bei jedem Druckvorgang das vorher auf dem Feld befindliche Zeichen gelöscht, d.h. überschrieben wird.
- TM kann den Lese-/Schreibkopf ein Feld nach links oder nach rechts bewegen.
- TM kann interne Zustände $0, 1, 2, 3, \ldots$ annehmen; 0 ist der Startzustand von TM.
- TM kann eine endliche Turing-Tafel (Turing-Programm) beobachten.

Turing-Tafel, Turing-Programm (1)

Definition A.1.2 (Turing-Tafel)

Eine Turing-Tafel T über einem (endlichen) Zeichenvorrat $\mathcal{A}$ ist eine Tafel mit 4 Spalten und $m + 1$ Zeilen, $m \geq 0$:

$$\begin{array}{llll}
i_0 & a_0 & b_0 & j_0 \\
i_1 & a_1 & b_1 & j_1 \\
\ldots & & & \\
i_k & a_k & b_k & j_k \\
\ldots & & & \\
i_m & a_m & b_m & j_m
\end{array}$$

Turing-Tafel, Turing-Programm (2)

Dabei bezeichnen in T:

- Das erste Element jeder Zeile den internen Zustand.
- Das zweite Element aus $\mathcal{A} \cup \{blank\}$ das unter dem Lese-/Schreibkopf liegende Zeichen.
- Das dritte Element b_k den Befehl "Drucke b_k", falls $b_k \in \mathcal{A} \cup \{blank\}$; den Befehl "Gehe nach links", falls $b_k = L$; den Befehl "Gehe nach rechts", falls $b_k = R$.
- Das vierte Element den internen Folgezustand aus $\mathbb{N}_0$.

wobei gilt:

- $i_k, j_k \in \mathbb{N}_0$.
- $a_k \in \mathcal{A} \cup \{blank\}$.
- $b_k \in \mathcal{A} \cup \{blank\} \cup \{L, R\}$, $L, R \notin \mathcal{A} \cup \{blank\}$.
- Weiters soll jedes Paar (i_k, a_k) höchstens einmal als Zeilenanfang vorkommen.

A.2

Markov-Algorithmen

Markov-Tafel

Definition A.2.1 (Markov-Tafel)

Eine Markov-Tafel T über einem (endlichen) Zeichenvorrat $\mathcal{A}$ ist eine Tafel mit 5 Spalten und $m+1$ Zeilen, $m \geq 0$:

0	a_0	i_0	b_0	j_0
1	a_1	i_1	b_1	j_1
...				
k	a_k	i_k	b_k	j_k
...				
m	a_m	i_m	b_m	j_m

Dabei gilt: $k \in [0..m]$, $a_k, b_k \in \mathcal{A}^*$, $\mathcal{A}^*$ Menge der Worte über $\mathcal{A}$ und $i_k, j_k \in \mathbb{N}_0$.

Markov-Algorithmus

Definition A.2.2 (Markov-Algorithmus)

Ein Markov-Algorithmus

$$M = (Y, Z, E, A, f_M)$$

ist gegeben durch

1. Eine Zwischenkonfigurationsmenge $Z = \mathcal{A}^* \times \mathbb{N}_0$.
2. Eine Eingabekonfigurationsmenge $E \subseteq \mathcal{A}^* \times \{0\}$.
3. Eine Ausgabekonfigurationsmenge $A \subseteq \mathcal{A}^* \times [m+1..\infty)$.
4. Eine Markov-Tafel T über $\mathcal{A}$ mit $m+1$ Zeilen und einer durch die Tafel T definierten (partiellen) Überführungsfunktion

   $$f_M : Z \to Z$$

   definiert durch:

Überführungsfunktion

$\forall\, x \in \mathcal{A}^*,\ \ k \in \mathbb{N}_0 :$

$$f_M(x,k) =_{df} \begin{cases} (x, i_k) & \text{falls } k \leq m \text{ und } a_k \text{ keine Teilzeichenreihe von } x \text{ ist.} \\ (\bar{x} b_k \bar{\bar{x}}, j_k) & \text{falls } k \leq m \text{ und } x = \bar{x} a_k \bar{\bar{x}} \text{, wobei die Länge von } \bar{x} \text{ minimal ist.} \\ \textit{undefiniert} & \text{falls } k > m. \end{cases}$$

A.3

Primitiv rekursive Funktionen

Primitiv rekursive Funktionen

Definition A.3.1 (Primitiv rekursive Funktionen)

Eine Funktion f heißt primitiv rekursiv, wenn f aus den Grundfunktionen $\lambda x.0$ und $\lambda x.x+1$ durch endlich viele Anwendungen expliziter Transformation, Komposition und primitiver Rekursion hervorgeht.

Transformation, Komposition

Definition A.3.2 (Explizite Transformation)

Eine Funktion g geht aus einer Funktion f durch explizite Transformation hervor, wenn es $e_1, \ldots, e_n$ gibt, so dass jedes e_i entweder eine Konstante aus $\mathbb{N}$ oder eine Variable x_i ist, so dass für alle $\bar{x}^m \in \mathbb{N}^m$ gilt:

$$g(x_1, \ldots, x_m) = f(e_1, \ldots, e_n)$$

Definition A.3.3 (Komposition)

Ist $f : \mathbb{N}^k \to \mathbb{N}_\perp$, $g_i : \mathbb{N}^n \to \mathbb{N}_\perp$ für $i = 1, \ldots, k$, dann ist $h : \mathbb{N}^k \to \mathbb{N}_\perp$ durch Komposition aus $f, g_1, \ldots, g_k$ definiert, genau dann wenn für alle $\bar{x}^n \in \mathbb{N}^n$ gilt:

$$h(\bar{x}^n) = \begin{cases} f(g_1(\bar{x}^n), \ldots, g_k(\bar{x}^n)) & \text{falls jedes } g_i(\bar{x}^n) \neq \perp \text{ ist} \\ \perp & \text{sonst} \end{cases}$$

Primitive Rekursion

Definition A.3.4 (Primitive Rekursion)

Ist $f : \mathbb{N}^n \rightarrow \mathbb{N}_\perp$ und $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}_\perp$, dann ist $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}_\perp$ durch primitive Rekursion definiert, genau dann wenn für alle $\bar{x}^n \in \mathbb{N}^n, t \in \mathbb{N}$ gilt:

$$h(0, \bar{x}^n) = f(\bar{x}^n)$$

$$h(t+1, \bar{x}^n) = \begin{cases} g(t, h(t, \bar{x}^n), \bar{x}^n) & \text{falls } h(t, \bar{x}^n) \neq \perp \\ \perp & \text{sonst} \end{cases}$$

A.4

μ-rekursive Funktionen

μ-rekursive Funktionen

Definition A.4.1 (μ-rekursive Funktionen)

Eine Funktion f heißt μ-rekursiv, wenn f aus den Grundfunktionen $\lambda x.0$ und $\lambda x.x+1$ durch endlich viele Anwendungen expliziter Transformation, Komposition, primitiver Rekursion und Minimierung totaler Funktionen hervorgeht.

Minimierung

Definition A.4.2 (Minimierung)

Ist $g : \mathbb{N}^{n+1} \to \mathbb{N}_\perp$, dann geht $h : \mathbb{N}^n \to \mathbb{N}_\perp$ aus g durch Minimierung hervor, genau dann wenn für alle $\bar{x}^n \in \mathbb{N}^n$ gilt:

$$h(\bar{x}^n) = \begin{cases} t & \text{falls } t \in \mathbb{N} \text{ die kleinste Zahl ist mit } g(t, \bar{x}^n) = 0 \\ \perp & \text{sonst} \end{cases}$$

A.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (1)

- Friedrich L. Bauer. *Historische Notizen – Wer erfand den von-Neumann-Rechner?* Informatik-Spektrum 21(3):84-89, 1998.
- Cristian S. Calude. *People and Ideas in Theoretical Computer Science*. Springer-V., 1999.
- Luca Cardelli. *Global Computation*. ACM SIGPLAN Notices 32(1):66-68, 1997.
- Gregory J. Chaitin. *The Limits of Mathematics*. Journal of Universal Computer Science 2(5):270-305, 1996.
- Gregory J. Chaitin. *The Limits of Mathematics – A Course on Information Theory and the Limits of Formal Reasoning*. Springer-V., 1998.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (2)

- Gregory J. Chaitin. *The Unknowable*. Springer-V., 1999.
- Paul Cockshott, Greg Michaelson. *Are There New Models of Computation? Reply to Wegner and Eberbach*. The Computer Journal 50(2):232-247, 2007.
- S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg). *New Computational Paradigms: Changing Conceptions of What is Computable*. Springer-V., 2008.
- B. Jack Copeland. *The Church-Turing Thesis*. The Stanford Encyclopedia of Philosophy, 2002. `http://plato.stanford.edu/entries/church-turing`

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (3)

- B. Jack Copeland. *Accelerating Turing Machines*. Minds and Machines 12(2):281-301, 2002.
- B. Jack Copeland. *Hypercomputation*. Minds and Machines 12(4):461-502, 2002.
- B. Jack Copeland, Eli Dresner, Diane Proudfoot, Oron Shagrir. *Viewpoint: Time to Reinspect the Foundations? Questioning if Computer Science is Outgrowing its Traditional Foundations*. Communications of the ACM 59(11):34-36, 2016.
- B. Jack Copeland, Carl J. Posy, Oron Shagrir (Hrsg.) *Computability: Turing, Gödel, Church, and Beyond.* MIT Press, 2013.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (4)

- Martin Davis. *What is a Computation?* Chapter in L.A. Steeb (Hrsg.), Mathematics Today – Twelve Informal Essays. Springer-V., 1978.
- Martin Davis. *Mathematical Logic and the Origin of Modern Computers*. Studies in the History of Mathematics, Mathematical Association of America, 137-165, 1987. Reprinted in: Rolf Herken (Hrsg.), The Universal Turing Machine – A Half-Century Survey, Kemmerer&Unverzagt und Oxford University Press, 149-174, 1988.
- Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. W.W. Norton and Company, 2000.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (5)

- Martin Davis. *The Myth of Hypercomputation*. Christof Teuscher (Hrsg.), Alan Turing: Life and Legacy of a Great Thinker, Springer-V., 195-212, 2004.
- Martin Davis. *The Church-Turing Thesis: Consensus and Opposition*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 125-132, 2006.
- Martin Davis. *Why There is No Such Discipline as Hypercomputation*. Applied Mathematics and Computation 178(1):4-7, Special issue on Hypercomputation, 2006.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (6)

- John W. Dawson Jr. *Gödel and the Origin of Computer Science*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 133-136, 2006.
- Peter J. Denning. *The Field of Programmers Myth*. Communications of the ACM 47(7):15-20, 2004.
- Peter J. Denning, Peter Wegner. *Introduction to What is Computation*. The Computer Journal 55(7):803-804, 2012.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (7)

- Charles E.M. Dunlop. Book review on: M. Gams, M. Paprzycki, X. Wu (Hrsg.). *Mind Versus Computer: Were Dreyfus and Winograd Right?*, Frontiers in Artificial Intelligence and Applications Vol. 43, IOS Press, 1997. Minds and Machines 10(2):289-296, 2000.
- Eugene Eberbach, Dina Q. Goldin, Peter Wegner. *Turing's Ideas and Models of Computation*. Christof Teuscher (Hrsg.), Alan Turing: Life and Legacy of a Great Thinker, Springer-V., 159-194, 2004.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (8)

- Bertil Ekdahl. *Interactive Computing does not Supersede Church's Thesis*. In Proceedings of the 17th International Conference on Computer Science, Association of Management and the International Association of Management, Vol. 17, No. 2, Part B, 261-265, 1999.
- Matjaž Gams. *The Turing Machine may not be the Universal Machine – A Reply to Dunlop*. Minds and Machines 12(1):137-142, 2002.
- Matjaž Gams. *Alan Turing, Turing Machines and Stronger*. Informatica 37(1):9-14, 2013.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (9)

- Dina Q. Goldin, Scott A. Smolka, Paul C. Attie, Elaine L. Sonderegger. *Turing Machines, Transition Systems, and Interaction*. Information and Computation Journal 194(2):101-128, 2004.
- Dina Q. Goldin, Peter Wegner. *The Interactive Nature of Computing: Refuting the Strong Church-Rosser Thesis*. Minds and Machines 18(1):17-38, 2008.
- Saul A. Kripke. *The Church-Turing "Thesis" as a Special Corollary of Gödel's Completeness Theorem*. In B. Jack Copeland, Carl J. Posy, Oron Shagrir (Hrsg.) Computability: Turing, Gödel, Church, and Beyond. MIT Press, 77-104, 2013.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (10)

- Michael Prasse, Peter Rittgen. *Bemerkungen zu Peter Wegners Ausführungen über Interaktion und Berechenbarkeit*. Informatik-Spektrum 21(3):141-146, 1998.
- Michael Prasse, Peter Rittgen. *Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability*. The Computer Journal 41(6):357-362, 1998.
- Edna E. Reiter, Clayton M. Johnson. *Limits of Computation: An Introduction to the Undecidable and the Intractable*. Chapman and Hall, 2012.
- Uwe Schöning. *Complexity Theory and Interaction*. In R. Herken (Hrsg.), The Universal Turing Machine – A Half-Century Survey. Springer-V., 1988.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (11)

- Jack T. Schwartz. *Do the Integers Exist? The Unknowability of Arithmetic Consistency*. Communications on Pure and Applied Mathematics 58:1280-1286, 2005.
- Wilfried Sieg. *Church without Dogma: Axioms for Computability*. In S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg.), New Computational Paradigms - Changing Conceptions of What is Computable, Springer-V., 139-152, 2008.
- Alan Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society 42(2):230-265, 1936. Correction, ibid, 43:544-546, 1937.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (12)

- Alan Turing. *Computing Machinery and Intelligence*. Mind 59:433-460, 1950.
- Jan van Leeuwen, Jirí Wiedermann. *On Algorithms and Interaction*. In Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS 2000), Springer-V., LNCS 1893, 99-112, 2000.
- Jan van Leeuwen, Jirí Wiedermann. *The Turing Machine Paradigm in Contemporary Computing*. In B. Enquist, W. Schmidt (Hrsg.), Mathematics Unlimited – 2001 and Beyond. Springer-V., 1139-1155, 2001.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (13)

- Jan van Leeuwen, Jirí Wiedermann. *Beyond the Turing Limit: Evolving Interactive Systems*. In Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2001), Springer-V., LNCS 2234, 90-109, 2001.
- Robin Milner. *Elements of Interaction: Turing Award Lecture*. Communications of the ACM 36(1):78-89, 1993.
- Hava T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit.* Birkhäuser, 1999.
- Peter Wegner. *Why Interaction is More Powerful Than Algorithms*. Communications of the ACM 40(5):81-91, 1997.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (14)

- Peter Wegner. *Interactive Foundations of Computing*. Theoretical Computer Science 192(2):315-351, 1998.
- Peter Wegner. *Observability and Empirical Computation*. The Monist 82(1), Issue on the Philosophy of Computation, 1999.
  www.cs.brown.edu/people/pw/papers/monist.ps
- Peter Wegner. *The Evolution of Computation*. The Computer Journal 55(7):811-813, 2012.
- Peter Wegner, Eugene Eberbach. *New Models of Computation*. The Computer Journal 47(1):4-9, 2004.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (15)

- Peter Wegner, Dina Q. Goldin. *Interaction, Computability, and Church's Thesis*. Accepted to the British Computer Journal.
  `www.cs.brown.edu/people/pw/papers/bcj1.pdf`
- Peter Wegner, Dina Q. Goldin. *Computation Beyond Turing Machines*. Communications of the ACM 46(4):100-102, 2003.
- Peter Wegner, Dina Q. Goldin. *The Church-Turing Thesis: Breaking the Myth*. In Proceedings of the 1st Conference on Computability in Europe – New Computational Paradigms (CiE 2005), Springer-V., LNCS 3526, 152-168, 2005.

B

Andere funktionale Sprachen

Schlaglichter

...auf ausgewählte andere funktionale Programmiersprachen und wesentliche ihrer Eigenschaften:

- ML: Starker Wettbewerber von Haskell mit sofortiger (engl. eager) Auswertung.
- Lisp: Der *Oldtimer* unter den funktionalen Sprachen.
- APL: Ein sprachlicher Exot.
- ...

ML: Eine Sprache mit ‘sofortiger’ Auswertung

ML, eine strikte funktionale Sprache.

Wichtige Eigenschaften:

- Starke Typisierung mit Typinferenz, keine Typklassen.
- Umfangreiches Typkonzept für Module und abstrakte Datentypen (ADTs).
- Lexical scoping, curryfizieren (wie Haskell).
- Zahlreiche Erweiterungen (z.B. in OCaml) auch für imperative und objektorientierte Programmierung.
- Sehr gute theoretische Fundierung.

ML-Programmbeispiel: Module/ADTs in ML

```
structure S = struct
    type 't Stack              = 't list;
    val  create                = Stack nil;
    fun  push x (Stack xs)     = Stack (x::xs);
    fun  pop (Stack nil)       = Stack nil;
    |    pop (Stack (x::xs))   = Stack xs;
    fun  top (Stack nil)       = nil;
    |    top (Stack (x:xs))    = x;
end;

signature st = sig type q; val push: 't -> q -> q; end;

structure S1:st = S;
```

Lisp: Der “Oldtimer” fkt. Programmierspr.

Lisp, eine bewährte und weiterhin häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen.

Wichtige Eigenschaften:

- Einfache, interpretierte Sprache, dynamisch typisiert.
- Listen sind gleichzeitig Daten und Funktionsanwendungen.
- Nur lesbar, wenn Programme gut strukturiert sind.
- Erfolgreicher Einsatz in vielen Bereichen, insbesondere künstliche Intelligenz, Expertensysteme.
- Umfangreiche Bibliotheken, leicht erweiterbar.
- Sehr gut zur Metaprogrammierung geeignet

Ausdrücke in Lisp

Beispiele für Symbole:	A	(Atom)
	austria	(Atom)
	68000	(Zahl)

Beispiele für Listen:
(plus a b)
((meat chicken) water)
(unc trw synapse ridge hp)
nil bzw. () entsprechen leerer Liste

Eine Zahl repräsentiert ihren Wert direkt —
ein Atom ist der Name eines assoziierten Werts.

(setq x (a b c)) bindet x global an (a b c)
(let ((x a) (y b)) e) bindet x lokal in e an a und y an b

Funktionen in Lisp

Das erste Element einer Liste wird normalerweise als Funktion interpretiert, anzuwenden auf die restlichen Listenelemente.

(quote a) bzw. 'a liefert Argument a selbst als Ergebnis.

Beispiele für primitive Funktionen:

```
(car '(a b c))     ->> a
(car 'a)           ->> error
(cdr '(a b c))     ->> (b c)
(cdr '(a))         ->> nil
(cons 'a '(b c))   ->> (a b c)
(cons '(a) '(b))   ->> ((a) b)
```

```
(atom 'a)          ->> t
(atom '(a))        ->> nil
(eq 'a 'a)         ->> t
(eq 'a 'b)         ->> nil
(cond ((eq 'x 'y) 'b)
      (t 'c))      ->> c
```

Funktionsdefinitionen in Lisp

- (lambda (x y) (plus x y)) ist Funktion mit zwei Parametern.
- ((lambda (x y) (plus x y)) 2 3) wendet diese Funktion auf die Argumente 2 und 3 an und liefert 5 als Resultat.
- (define (add (lambda (x y) (plus x y)))) definiert einen globalen Namen "'add'" für die Funktion.
- (defun add (x y) (plus x y)) ist abgekürzte Schreibweise dafür.

Beispiel:

```
(defun reverse (l) (rev nil l))
(defun rev (out in)
     (cond ((null in) out)
             (t (rev (cons (car in) out) (cdr in)))))
```

Closures in Lisp

- Kein curryfizieren in Lisp, sog. closures als Ersatz.
- Closures: lokale Bindungen behalten Wert auch nach Verlassen der Funktion.

  Beispiel:

  ```
  (let ((x 5))
    (setf (symbol-function 'test)
      #'(lambda () x)))
  ```

- Praktisch: Funktion gibt closure zurück.

  Beispiel:

  ```
  (defun create-function (x)
    (function (lambda (y) (add x y))))
  ```

- Closures sind flexibel, aber curryfizieren ist viel einfacher.

Dynamisches vs. statisches Binden

...engl. dynamic scoping, static scoping.

- Lexikalisch: Bindung ortsabhängig (Quellcode).
- Dynamisch: Bindung vom Zeitpunkt abhängig.
- 'Normales' Lisp: Lexikalisches Binden.

  Beispiel:
  ```
  (setq a 100)
  (defun test () a)
  (let ((a 4)) (test))
  ```
  $\Rightarrow$ 100
- Dynamisches Binden durch (defvar a) möglich.

  Das obige Beispiel liefert damit 4.

Makros

- Code expandiert, nicht als Funktion aufgerufen (wie C).
- Definition: Erzeugt Code, der danach evaluiert wird.

  Beispiel:
  ```
  (defmacro get-name (x n)
      (list 'cadr (list 'assoc x n)))
  ```
- Expansion und Ausführung:

  ```
  (get-name 'a b) <<->> (cadr (assoc 'a b))
  ```
- Nur Expansion:

  ```
  (macroexpand '(get-name 'a b)) ->> '(cadr (assoc 'a b))
  ```

Lisp im Vergleich mit Haskell

Kriterium	Lisp	Haskell
Basis	Einfacher Interpreter	Formale Grundlage
Zielsetzung	Viele Bereiche	Referentiell transparent
Verwendung	Noch häufig	Zunehmend
Sprachumfang	Riesig (kleiner Kern)	Moderat, wachsend
Syntax	Einfach, verwirrend	Modern, Eigenheiten
Interaktivität	Hervorragend	Mit Einschränkungen
Typisierung	Dynamisch, einfach	Statisch, modern
Effizienz	Relativ gut	Relativ gut
Zukunft	Noch lange genutzt	Einflussreich

APL: Ein Exot unter den Sprachen

APL, eine ältere applikative (funktionale) Sprache mit imperativen Zusätzen.

Wichtige Eigenschaften:

- Dynamische Typisierung.
- Verwendung speziellen Zeichensatzes.
- Zahlreiche Funktionen (höherer Ordnung) sind vordefiniert; Sprache aber nicht einfach erweiterbar.
- Programme sehr kurz und kompakt, aber kaum lesbar.
- Besonders für Berechnungen mit Feldern gut geeignet.

APL-Programmentwicklung

...anhand eines Beispiels: Berechne d. Primzahlen von 1 bis N:

Schritt 1. $(\iota N) \circ.| (\iota N)$

Schritt 2. $0 = (\iota N) \circ.| (\iota N)$

Schritt 3. $+/[2]\ 0 = (\iota N) \circ.| (\iota N)$

Schritt 4. $2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))$

Schritt 5. $(2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N)))\ /\ \iota N$

C

Datentypdeklarationen in Pascal

Aufzählungstypen in Pascal

```
TYPE jahreszeiten = (fruehling, sommer, herbst, winter);
     spielfarbe   = (karo, herz, pik, kreuz);
     werktage     = (montag, dienstag, mittwoch,
                     donnerstag, freitag);
     wochenende   = (samstag, sonntag);
```

Bemerkung:

- Gleichheits- und Ordnungsrelationen sind auf Aufzählungstypen automatisch definiert (entspricht `deriving (Eq,Ord)`), so dass Aufzählungstypwerte verglichen werden können, z.B. karo $=$ pik $\rightsquigarrow$ false, karo $<$ pik $\rightsquigarrow$ true, kreuz $>=$ herz $\rightsquigarrow$ true, herz $<>$ kreuz $\rightsquigarrow$ true.
- Die Funktionen succ und pred liefern den Nachfolge- und Vorgängerwert eines Werts, die Funktion ord seine Position in der Aufzählung (entspricht `deriving Enum`), z.B. succ (herz) $\rightsquigarrow$ pik, pred (herz) $\rightsquigarrow$ karo, succ (kreuz) undef., ord (karo) $\rightsquigarrow$ 0, ord (kreuz) $\rightsquigarrow$ 3.

Produkttypen in Pascal

```
TYPE person    = RECORD
                  name: ARRAY [1..50] OF char;
                  geschlecht: (maennlich, weiblich);
                  alter: 0..150
                 END;

     anschrift = RECORD
                  gemeinde: ARRAY [1..50] OF char;
                  strasse: ARRAY [1..75] OF char;
                  hausnr: integer;
                  land: ARRAY [1..100] OF char
                 END;
```

Bemerkung:

- Der Typ von alter ist hier als Ausschnittstyp ganzer Zahlen definiert. Werte des Typs 0..150 sind die Zahlen von 0 bis 150.
- Bereichsüberschreitungen zur Laufzeit werden automatisch überprüft und führen zum Programmabbruch.

Summentypen in Pascal

```
TYPE index1 = 1..5;
TYPE index2 = 1..100;
TYPE traegermedium = (buch, ebuch, dvd, cd);
TYPE bildSchriftUndTonTraeger =
 RECORD
  CASE
   medium: traegermedium OF
   buch: (autor, titel, verlag: ARRAY [index2] OF char;
          auflage: 1..20; lieferbar: boolean);
   ebuch: (autor, titel, verlag: ARRAY [index2] OF char;
           lizenzBisJahr: integer);
   dvd: (titel, regisseur: ARRAY [index2] OF char;
         hauptdarsteller,sprachen: ARRAY [index1,index2]
                                            OF char);
   cd: (kuenstler, titel: ARRAY [index2] OF char;
        spieldauer: ARRAY [1..3] OF integer)
 END;
```

Mengentypen in Pascal

```
TYPE buchstaben = 'a'..'z';
TYPE zutaten = (mehl, zucker, salz, hefe, eier, essig,
                honig, rosinen, mandeln, joghurt, obst)

TYPE buchstabensuppe = SET OF buchstaben;
TYPE rezept = SET OF zutaten;

VAR vokalsuppe, allerleisuppe: buchstabensuppe;
VAR lebkuchen, nachtisch, verdorben: rezept;

vokalsuppe     := ['a','o','e','u']
                    * ['u','a'..'g'];     (Durchschnitt)
allerleisuppe := ['a'..'z'] - vokalsuppe;   (Differenz)
lebkuchen      := [mehl..salz,eier,honig..mandeln];
nachtisch      := [joghurt,obst];
verdorben      := lebkuchen + [essig];    (Vereinigung)
```

Bemerkung: Mengentypen in Pascal besitzen Eigenschaften und Funktionen, die denen von Listentypen und automatischer Listengenerierung in funktionalen Sprachen ähneln.

D

Hinweise zur schriftlichen Prüfung

Hinweise zur schriftlichen LVA-Prüfung (1)

- Worüber:
  - Vorlesungs- und Übungsstoff.
  - Folgender wissenschaftlicher (Übersichts-) Artikel:

    John W. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Communications of the ACM 21(8):613-641, 1978.
    (Zugänglich aus TUW-Netz in ACM Digital Library: `http://dl.acm.org/citation.cfm?id=359579`)
- Wann, wo, wie lange:
  - Haupttermin: Vorauss. am
    - **Do, den 18.01.2018,** 16:00 Uhr s.t. bis ca. 18:00 Uhr, Hörsaal EI7 (und ggf. EI3), Gußhausstr. 25-29; die Dauer beträgt 90 Minuten.
- Hilfsmittel: **Keine.**

Hinweise zur schriftlichen LVA-Prüfung (2)

- Anmeldung:
  - Ist erforderlich!
    - Wann: Von vorauss. Mo, 11.12.2017 (01:00 Uhr) bis vorauss. Mo, 15.01.2018 (12:00 Uhr).
    - Wie: Elektronisch über TISS.
- Mitzubringen:
  - **Studierendenausweis, Stift** (Papier wird gestellt).
- Voraussetzung:
  - Mindestens 50% der Punkte aus dem Übungsteil.
- Wichtig:
  - Verbindlich sind im Zweifel allein die in TISS angegebenen Termine, Fristen und Räume.

Hinweise zur schriftlichen LVA-Prüfung (3)

- Neben dem Haupttermin wird es drei Nebentermine für die schriftliche LVA-Prüfung geben, und zwar:
  - zu Anfang
  - in der Mitte
  - am Ende

  der Vorlesungszeit im SS 2018. Zeugnisausstellung stets zum frühestmöglichen Zeitpunkt; insbesondere nach jedem Klausurantritt; spätestens nach Ablauf des letzten Termins für die schriftliche Prüfung.

- Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung in TISS zwingend erforderlich.

- Die genauen Termine werden in TISS angekündigt!