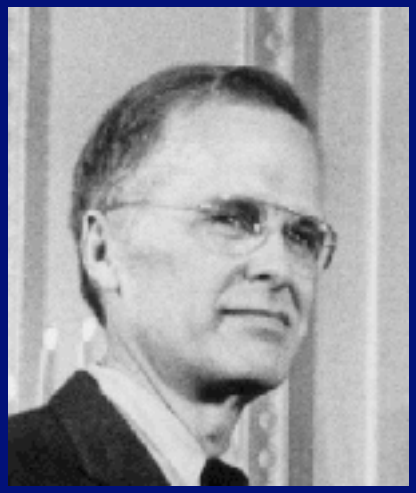
Funktionale Programmierung mit Haskell

Jan Hermanns

Programmiersprachen

imperativ		deklarativ	
konventionell	OO	logisch	funktional
Fortran	Smalltalk	Prolog	Lisp
C	Eiffel		ML
Pascal	Java		Haskell

von Neumann Flaschenhals



Not only is this tube a literal bottleneck for the data traffic problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand.

- John Backus, 1978

Haskell

- Benannt nach Haskell Brooks Curry
- Hervorgegangen aus Miranda
  - 1988 erste Definition des Haskell-Standards
  - aktueller Standard Haskell98
- Besondere Eigenschaften
  - Unterstützt “Literate-Programming”
  - polymorphes Typsystem (samt Inferenzsystem!)
  - lazy
  - rein funktional

Was bedeutet das konkret?

- Es gibt weder Variablen noch Zuweisungen
  - `int i=0; i=i+1;` geht nicht!
- Es gibt keine Schleifenkonstrukte
  - `for (int i=0; i<x; i++)` gibt es nicht!
  - `while (true) {}` gibt es auch nicht!
- Keine sequentielle Abarbeitung des Programm-Codes
- D.h. Umdenken ist angesagt!

Was macht dieses Programm?

```
qs( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qs( a, lo, l-1 );
    qs( a, l+1, hi );
  }
}
```

```
template <typename T>
void qsort (T *result, T *list, int n) {
    if (n == 0) return;
    T *smallerList, *largerList;
    smallerList = new T[n];
    largerList = new T[n];
    T pivot = list[0];
    int numSmaller=0, numLarger=0;
    for (int i = 1; i < n; i++)
        if (list[i] < pivot)
            smallerList[numSmaller++] = list[i];
        else
            largerList[numLarger++] = list[i];

    qsort(smallerList,smallerList,numSmaller);
    qsort(largerList,largerList,numLarger);

    int pos = 0;
    for ( int i = 0; i < numSmaller; i++)
        result[pos++] = smallerList[i];

    result[pos++] = pivot;

    for ( int i = 0; i < numLarger; i++)
        result[pos++] = largerList[i];

    delete [] smallerList;
    delete [] largerList;
}
```

und in Haskell ...

```
qsort []     = []
qsort (x:xs) = qsort lessX ++ [x] ++ qsort greaterEqX
               where
                 lessX      = [e | e<-xs, e < x]
                 greaterEqX = [e | e<-xs, e >= x]
```

Einfache Funktionen

- Namen von Funktionen und Parametern müssen mit Kleinbuchstaben beginnen

Beispiele:

```
hello = “Hello, world!”

square x = x * x

add x y = x + y
```

Funktionen mit Guards

- Ein Guard ist ein “bewachter Ausdruck”
- Abarbeitung der Guards von oben nach unten
- Der Ausdruck hinter dem ersten “passenden” Guard wird ausgewertet

```
maximum x y
        | x >= y     = x
        | otherwise = y
```

Rekursive Funktionen

```
-- Bildet die Summe 1+2+...+n.

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

```
sumUp 2
2 + sumUp (2-1)
2 + 1 + sumUp (1-1)
2 + 1 + 0
3
```

```
sumUp (-2)
(-2) + sumUp (-3)
(-2) + (-3) + sumUp (-4)
usw.
```

Die error Funktion

- Aufruf von sumUp mit negativen Werten führt zu Endlos-Rekursion!
- Eine bessere Definition wäre daher

```
-- Bildet die Summe 1+2+...+n.
-- Parameter n darf nicht negativ sein

sumUp n
      | n == 0     = 0
      | n > 0      = n + sumUp (n-1)
      | otherwise = error “n is negative!”
```

Lokale Definitionen

- Mit dem Schlüsselwort where kann man lokale Definitionen vornehmen
- Erhöht die Lesbarkeit

```
-- Summiert die Quadrate von x und y.

sumSquares x y = square x + square y
      where
        square n = n * n
```

Operatoren

Operatoren wie +, *, -, / sind normale Funktionen, die zwei Parameter erwarten. Mit den Zeichen !#$%&*+./<=>?\^|:-~ können eigene Operatoren definiert werden.

```
x +* y = (x + y) * (x + y)
2 +* 3 → 25
```

Prefix-Schreibweise ebenfalls möglich.

```
(+) 3 4 → 7
```

Jede “2-argumentige” Funktionen kann mittels Backquotes als Operator verwendet werden.

```
2 `max` 3 → 3
```

Layout-Regeln

- Blöcke werden durch { und } getrennt
- Einzelne Definitionen werden durch , getrennt
- Bei Verwendung von Einrückung kann jedoch auf die Trennsymbole verzichtet werden.
- vgl. Quicksort Definition

Typen

- Haskell ist stark getypt
  - d.h. zur Laufzeit können keine Fehler durch Anwendung von Funktionen auf Argumente des falschen Typs entstehen
- Haskell hat ein statisches Typsystem
  - die Typen aller Ausdrücke sind zur Übersetzungszeit bekannt
  - sie werden sogar automatisch inferiert
- Hakell ist polymorph
  - d.h. Funktionen können nicht nur für einzelne Typen, sondern auch für ganze Typklassen definiert werden

Typdeklarationen

- Typen werden mit Hilfe der Symbole :: und -> definiert.
- Vordefinierte Basistypen sind u.a. Bool, Int, Integer, Float, Double, Char
- Typnamen beginnen mit Grossbuchstaben

```
daysInWeek :: Int
daysInWeek = 7

square :: Int -> Int
square x = x * x

add :: Int -> Int -> Int
add x y = x + y
```

Typsynonyme

- Mit dem Schlüsselwort type werden Typsynonyme definiert

```
type Age = Int

isAdult :: Age -> Bool
isAdult x = x >= 18
```

Listen

- Listen sind die zentralen Datenstrukturen in funktionalen Programmen
- Listen sind Folgen von Werten des gleichen Typs. z.B. [Bool], [Int], [[Char]]
- Manche Listenfunktionen haben einen eindeutigen Typ
  - sum :: [Int] -> Int
- Für andere Listenfunktionen ist der Typ der Elemente jedoch irrelevant!?
  - length [1, 2, 3]
  - length [‘a’, ‘b’, ‘c’]

Typvariablen

- Mit Hilfe von Typvariablen wird ausgedrückt, daß kein bestimmter Typ erwartet wird.
  - generischer Polymorphismus
- Für Typvariablen werden üblicherweise Kleinbuchstaben verwendet

```
length :: [a] -> Int

id :: a -> a
id x = x
```

Konstruktion von Listen

- [] ist die leere Liste
- Mittels : wird ein Element am Anfang der Liste angefügt
  - Der Typ von : ist a -> [a] -> [a]
  - 1 : [] ergibt [1]
  - 1:(2:[]) ergibt [1,2]
- : ist rechts-assoziativ
  - 1:(2:(3:[])) → 1:2:3:[]

Syntaktischer Zucker

- [1, 2, 3] ≈ 1:2:3:[]
- Strings sind ganz normale Listen
  - type String = [Char]
  - “abc” → [‘a’, ‘b’, ‘c’]
- Aufzählungen
  - [1 .. 4] → [1, 2, 3, 4]
  - [1, 3 .. 10] → [1, 3, 5, 7, 9]
  - [‘a’, ‘c’ .. ‘n’] → "acegikm"
  - [1 ..] die Liste von 1 bis ∞

Pattern Matching

Funktionsdefinitionen können “auseinander gezogen” werden.

Abhängig vom übergebenen Parameter wird dann die entsprechende Definition angewendet.

```
not       :: Bool -> Bool
not True  = False
not False = True

not       :: Bool -> Bool
not x
      | x           = False
      | otherwise   = True
```

Wildcards

Das funktioniert auch für mehrere Parameter.

```
and :: Bool -> Bool -> Bool
and True  True  = True
and False True  = False
and True  False = False
and False False = False
```

Mit Hilfe des Wildcard-Patterns _ kann die obige Definition vereinfacht werden.

```
and :: Bool -> Bool -> Bool
and True True = True
and _ _       = False
```

List Patterns

Eine Liste von Patterns kann selbst auch als Pattern verwendet werden.

```
foo :: String -> Bool
foo ['a', _, _] = True
foo _           = False
```

Das gilt auch für die leere Liste.

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Mit dem cons Operator : können nicht nur Listen, sondern auch Patterns gebaut werden.

```
foo :: String -> Bool
foo ('a':_) = True
foo _       = False
```

Die Variablen eines Patterns werden bei einem Match an die übergebenen Werte gebunden.

```
startsWith :: String -> Char -> Bool
startsWith [] _ = False
startsWith (x:_) y
        | x == y    = True
        | otherwise = False


"Hello" `startsWith` 'H' → True
```

Jeder Parameter einer Funktionsdefinition muss einen eigenen, eindeutigen Namen tragen.

Daher ist die folgende Definition leider nicht gültig:

```
startsWith :: String -> Char -> Bool
startsWith [] _     = False
startsWith (x:_) x = True
```

```
ERROR - Repeated variable "x" in pattern
```

Standard Listenfunktionen

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

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

last              :: [a] -> a
last [x]           = x
last (_:xs)        = last xs

init              :: [a] -> [a]
init [x]           = []
init (x:xs)        = x : init xs

(++)              :: [a] -> [a] -> [a]
[]     ++ ys       = ys
(x:xs) ++ ys       = x : (xs ++ ys)
```

Die Funktion last liefert das letzte Element einer Liste.

```
     last              :: [a] -> a
(1)  last [x]           = x
(2)  last (_:xs)        = last xs
```

```
last [1,2,3] (2)

last [2,3]   (2)

last [3]     (1)

3
```

Die Funktion init liefert die Liste ohne das letzte Element.

```
     init                :: [a] -> [a]
(1)  init [x]             = []
(2)  init (x:xs)          = x : init xs
```

```
init [1,2,3]      (2)
1 : init [2,3]    (2)
1 : 2 : init [3]  (1)
1 : 2 : [] → [1, 2]
```

Der Operator ++ hängt zwei Listen aneinander.

```
      (++)                :: [a] -> [a] -> [a]
(1)   []      ++ ys        = ys
(2)   (x:xs)  ++ ys        = x : (xs ++ ys)
```

```
[1,2,3] ++ [4,5]                      (2)
1 : ([2,3] ++ [4,5])                  (2)
1 : (2 : ([3] ++ [4,5]))              (2)
1 : (2 : (3 : ([] ++ [4,5])))         (1)
1 : (2 : (3 : ([4,5])))
1 : (2 : ([3,4,5]))
1 : ([2,3,4,5])
([1,2,3,4,5])
[1,2,3,4,5]
```

Tuppel

- Sequenzen mit fester Länge
- Die Elemente können unterschiedliche Typen haben
- Vergleichbar mit C-structs oder Pascal-records

```
(False, True)

(“Jan”, “Hermanns”, 29)

((1, ‘a’), [1, 2, 3])

([(1, ‘a’), (2, ‘b’)], 4711, “foo”)
```

Mit Hilfe von Tuppeln und dem Schlüsselwort type kann man eigene Datenstrukturen definieren.

```
type Name   = String
type Age    = Int
type Person = (Name, Age)

createPerson :: Name -> Age -> Person
createPerson n a = (n, a)

getAge :: Person -> Age
getAge (_, a) = a

getName :: Person -> Name
getName (n, _) = n
```

Für Paare (2-Tuppel) und Trippel (3-Tuppel) gibt es ein paar vordefinierte Funktionen.

Daher könnte man die Definitionen:

```
getAge :: Person -> Age
getAge (_, a) = a

getName :: Person -> Name
getName (n, _) = n
```

wie folgt vereinfachen:

```
getAge :: Person -> Age
getAge p = snd p

getName :: Person -> Name
getName p = fst p
```

List Comprehensions

In Anlehnung an die Schreibweise aus der Mengenlehre,

$\{x^2 \mid x \in \{1 \ldots \infty\}\}$

können Listen auch in Haskell definiert werden.

```
[x^2 | x <- [1..]]
 → [1,4,9,16,25,36,49,64,81,usw.]

[(x, y) | x <- [1..3], y <- [1..2]]
 → [(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

[(x, y) | x <- [1..3], y <- [1..x]]
 → [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

Innerhalb einer List Comprehension können tests definiert werden.

```
[x | x <- [2,4..10]]
 → [2,4,6,8,10]

[x | x <- [1..10], even x]
 → [2,4,6,8,10]

[x | x <- [1..10], even x, x>3]
 → [4,6,8,10]
```

Auf Basis der bereits definierten Funktionen:

```
getAdults   :: [Person] -> [Person]
getAdults ps = [p | p <- ps, isAdult (getAge p)]
```

Quicksort revisited

```
qs []     = []
qs (x:xs) = qs lessX ++ [x] ++ qs greaterEqX
            where
              lessX      = [e | e<-xs, e < x]
              greaterEqX = [e | e<-xs, e >= x]
```

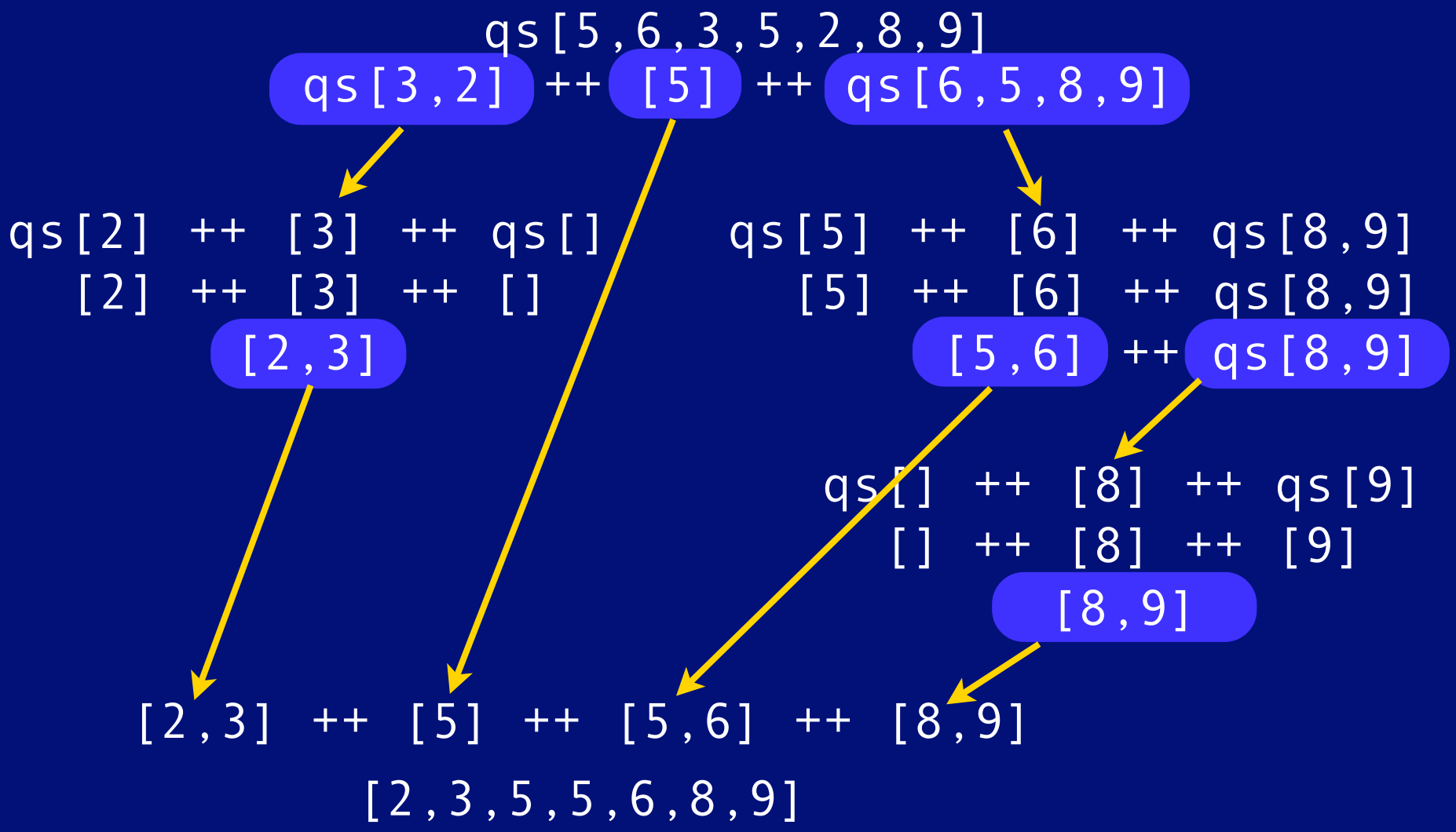
Folgende Eigenschaften sollten klar sein:

```
qs[] → []

qs[4]
qs[] ++ [4] ++ qs[]
[]   ++ [4] ++ []
[4]

qs[4711] → [4711]
```

```
qs []     = []
qs (x:xs) = qs lessX ++ [x] ++ qs greaterEqX
            where
              lessX      = [e | e<-xs, e < x]
              greaterEqX = [e | e<-xs, e >= x]
```



Higher-order functions

Funktionen können als Daten-Objekte an andere Funktion übergeben oder als Werte zurückgeliefert werden.

```
map           :: (a -> b) -> [a] -> [b]
map f xs       = [ f x | x <- xs ]

double        :: Int -> Int
double x       = 2 * x

doubleAll     :: [Int] -> [Int]
doubleAll xs = map double xs
```

Lambda Ausdrücke

Mit Hilfe von λ –Ausdrücken können “anonyme” Funktionen erzeugt werden.

```
doubleAll          :: [Int] -> [Int]
doubleAll xs        = map (\x -> x*2) xs
```

λ –Ausdrücke können auch mehrere Parameter haben.

```
applyToBoth :: (a -> a -> a) -> a -> a
applyToBoth f x = f x x

applyToBoth (\x y -> x+y) 5 → 10
```

Listen kumulieren

```
     foldr1              :: (a -> a -> a) -> [a] -> a
(1)  foldr1 f [x]         = x
(2)  foldr1 f (x:xs)      = f x (foldr1 f xs)
```

```
foldr1 (+) [1..4]                          (2)
(+) 1 (foldr1 (+) [2,3,4])                 (2)
(+) 1 ((+) 2 (foldr1 (+) [3,4]))           (2)
(+) 1 ((+) 2 ((+) 3 (foldr1 (+) [4])))     (1)
(+) 1 ((+) 2 ((+) 3 (4)))
(+) 1 ((+) 2 (7))
(+) 1 (9)
10
```

```
sumUp n = foldr1 (+) [0..n]
```

foldr1 funktioniert nur für Listen mit mindestens einem Element.

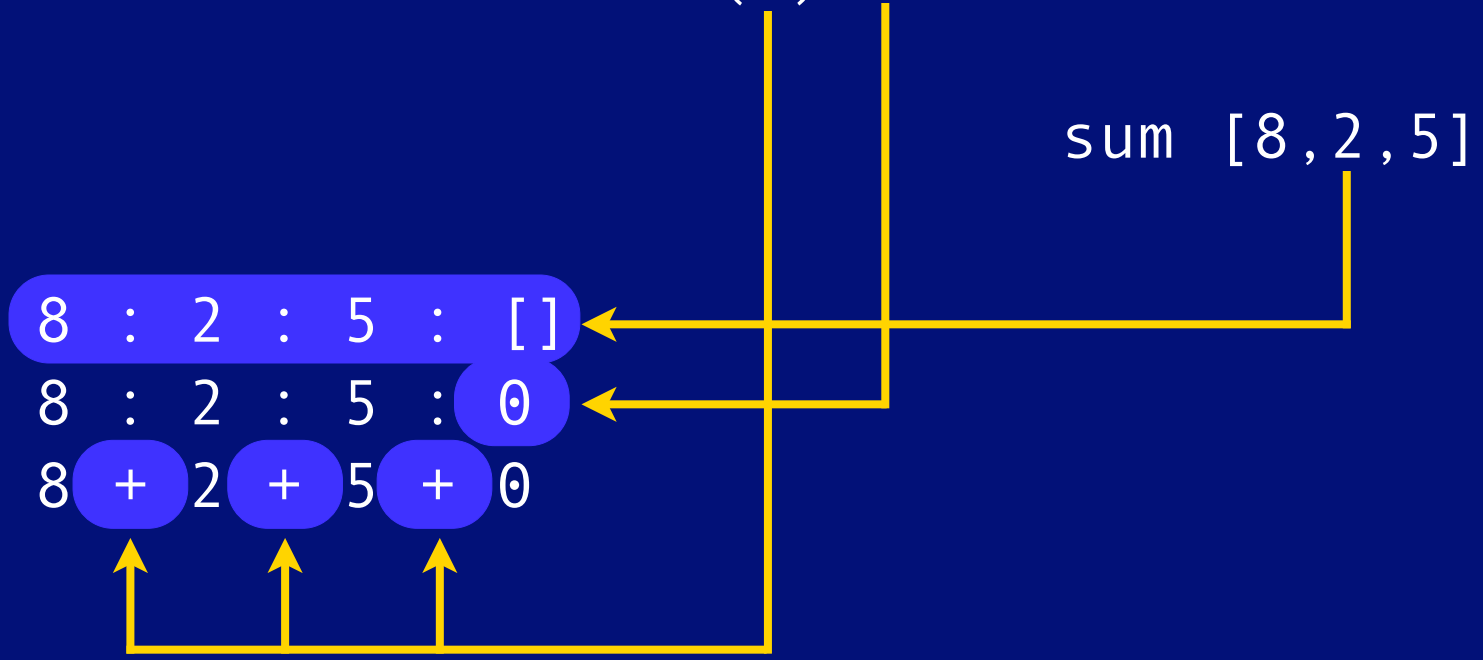
```
foldr                :: (a -> b -> b) -> b -> [a] -> b
(1) foldr f z []        = z
(2) foldr f z (x:xs)    = f x (foldr f z xs)

len                  :: [a] -> Int
len xs                = foldr (\_ n -> n+1) 0 xs

len “Jan”
foldr (\_ n -> n+1) 0 [‘J’,’a’,’n’]     (2)
λ ‘J’ (foldr λ 0 [‘a’,’n’])             (2)
λ ‘J’ (λ ‘a’ (foldr λ 0 [’n’]))         (2)
λ ‘J’ (λ ‘a’ (λ ‘n’ (foldr λ 0 [])))    (1)
λ ‘J’ (λ ‘a’ ((\_ n -> n+1) ‘n’ 0))
λ ‘J’ ((\_ n -> n+1) ‘a’ 1)
(\_ n -> n+1) ‘J’ 2
3
```

foldr tauscht den cons Operator durch die übergebene Funktion aus und die leere Liste durch das initiale Element.

```
sum    :: [Int] -> Int
sum xs = foldr (+) 0 xs
```




```
8 + (2 + (5 + 0))
```

Durch die Verwendung von Higher-order functions kann man Definitionen sehr abstrakt formulieren.

```
overallAge   :: [Person] -> Int
overallAge ps = foldr (+) 0 (map getAge ps)
```

Partielle Auswertung

Das -> Symbol ist rechts-assoziativ, so dass die Typdeklarationen

```
f :: a -> b -> c
g :: a -> b -> c -> d
h :: a -> b -> c -> d -> e
```

interpretiert werden als

```
f :: a -> (b -> c)
g :: a -> (b -> (c -> d))
h :: a -> (b -> (c -> (d -> e)))
```

Funktionen liegen implizit in curried Form vor.

Die Anwendung einer Funktion auf ihre Argumente ist links-assoziativ.

```
mul     :: Int -> Int -> Int
mul x y = x * y

mul 3 5
```

Die Anwendung passiert daher schrittweise von links nach rechts.

```
(mul 3) 5
```

Beispiele

Die folgenden Funktionen

```
mul    :: Int -> Int -> Int

map    :: (a -> b) -> [a] -> [b]

foldr  :: (a -> b -> b) -> b -> [a] -> b
```

haben die impliziten Typen

```
mul    :: Int -> (Int -> Int)

map    :: (a -> b) -> ([a] -> [b])

foldr  :: (a -> (b -> b)) -> (b -> ([a] -> b))
```

Übergibt man einer Funktion weniger Parameter als “vorgesehen”, so erhält man eine partiell ausgewertete Funktion zurück.

```
mul         :: Int -> (Int -> Int)
mul x y      = x * y

double      :: Int -> Int
double       = mul 2

double 4 → 8
```

Partiell ausgewertete Funktionen können ganz normal als Daten übergeben werden.

```
doubleAll    :: [Int] -> [Int]
doubleAll     = map (mul 2)
```

Der implizite Typ von foldr.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> (b -> ([a] -> b))
foldr :: (a -> (b -> b)) -> (b -> ([a] -> b))
```

Das Currying funktioniert auch mit mehreren Parametern.

```
concat              :: [[a]] -> [a]
concat               = foldr (++) []

and, or             :: [Bool] -> Bool
and                  = foldr (&&) True
or                   = foldr (||) False
```

Operator sections

Partielle Auswertung funktioniert natürlich auch bei Operatoren.

```
(op x) y = y op x
(x op) y = x op y
```

Beispiele:

```
(+2)        -- addiert 2 zum Argument
(2+)        -- addiert 2 zum Argument
(>2)        -- prueft, ob Argument groesser 2
(3:)        -- haengt 3 an den Kopf einer Liste
(++"\n")    -- Newline am Ende des Strings
("\n"++)    -- Newline am Anfang des Strings

doubleAll = map (2*)
```

Schlussbemerkungen

- Haskell hat noch wesentlich mehr zu bieten
  - Information-Hiding (Module)
  - algebraische Datentypen
  - einschränkenden Polymorphismus (Typklassen)
  - ein interessantes I/O-Konzept (Monaden)

Weitere Informationen finden Sie unter:

http://www.haskell.org

Sehr empfehlenswert ist das Buch:

"Haskell - The Craft of Functional Programming"
von Simon Thompson
ISBN 0-201-34275-8