Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2016/2017

(Stand: 11.10.2016)

Jens Knoop

Technische Universität Wien
Institut für Computersprachen

Inhaltsverzeichnis

Inhaltsverzeichnis (1)

Teil I: Einführung

Inhaltsverzeichnis (2)

Inhaltsverzeichnis (3)

Inhaltsverzeichnis (4)

Teil III: Funktionale Programmierung

Inhaltsverzeichnis (5)

Inhaltsverzeichnis (6)

Teil V: Ergänzungen und weiterführende Konzepte

Inhaltsverzeichnis (7)

Inhaltsverzeichnis (8)

Inhaltsverzeichnis (9)

Teil I

Einführung

Kapitel 1

Motivation

Überblick

Funktionale Programmierung, funktionale Programmierung in Haskell

Beachte: 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

Beispiele – Die ersten Zehn

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 eine Ein-/Ausgabeoperation.

Interessant, jedoch nicht selbsterklärend: Der Typ der Funktion `putStrLn`

```
putStrLn :: String -> IO ()
```

Aber: Auch die Java-Entsprechung

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

...bedarf einer weiter ausholenden Erklärung.

2) Fakultätsfunktion

$$! : I\!N \rightarrow I\!N$$

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

2) Fakultätsfunktion (fgs.)

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

...ein Beispiel für eine rekursive Funktionsdefinition.

Aufrufe:

```
fac 4 ->>  24
fac 5 ->> 120
fac 6 ->> 720
```

Lies: *"Die Auswertung des Ausdrucks/Aufrufs $fac\ 4$ liefert den Wert 24; der Ausdruck/Aufruf $fac\ 4$ hat den Wert 24."*

2) Fakultätsfunktion (fgs.)

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

2) Fakultätsfunktion (fgs.)


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

2) Fakultätsfunktion (fgs.)

```
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | n > 0     = n * fac (n - 1)
  | otherwise = error "fac: Nur positive Argumente!"
```

...ein Beispiel für eine einfache Form der Fehlerbehandlung.

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

Konzeptuell

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 noch nicht gestrichenen Zahl.

Illustration

```
Schritt 1:
 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17...
Schritt 2 (Streichen der Vielfachen von "2"):
 2  3     5     7     9     11    13    15    17...
Schritt 2 (Streichen der Vielfachen von "3"):
 2  3     5     7           11    13          17...
Schritt 2 (Streichen der Vielfachen von "5"):...
```

3) Das Sieb des Eratosthenes (fgs.)

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

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

...ein Beispiel für die Programmierung mit Strömen.

Aufrufe:

```
primes ->> [2,3,5,7,11,13,17,19,23,29,31,37,41,...
take 10 primes ->> [2,3,5,7,11,13,17,19,23,29]
```

4) Binomialkoeffizienten

Die Anzahl der Kombinationen k-ter Ordnung von n Elementen ohne Wiederholung.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
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 (fgs.)

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 (kaskadenartig-) rekursive Funktionsdefinition.

Aufrufe:

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

4) Binomialkoeffizienten (fgs.)

Uncurryfizierte Darstellung:

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

Curryfizierte Darstellung:

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

Aufrufe:

```
binom (49,6) ->> 13.983.816
binom (45,6) ->>  8.145.060
binomC 49 6 ->> 13.983.816
binomC 45 6 ->>  8.145.060
binomC 49 :: Integer -> Integer
   (binomC 49) bezeichnet die Funktion "49über"
```

5) Umkehren einer Zeichenreihe

```
type String = [Char]

reverse :: String -> String
reverse ""     = ""
reverse (c:cs) = (reverse cs) ++ [c]
```

...ein Beispiel für eine Funktion auf Zeichenreihen.

Aufrufe:

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

6) Reißverschlussfunktion

...zum ‘Verschmelzen’ zweier Listen zu einer Liste von Paaren.

```
zip :: [a] -> [b] -> [(a,b)]
zip _ []          = []
zip [] _          = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

...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] []           ->> []
```

7) Addition

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

...ein Beispiel für eine überladene Funktion.

Aufrufe:

```
(+) 2 3      ->> 5
2 + 3        ->> 5
(+) 2.1 1.04 ->> 3.14
2.1 + 1.04   ->> 3.14
(+) 2.14 1   ->> 3.14 (automatische Typanpassung)
(+) 1 :: Integer -> Integer  (Inkrementfunktion)
```

8) Die map-Funktion

```
map :: (a->b) -> [a] -> [b]
map f [] = []
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 ($m \geq 0$, $n > 0$).

```
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
```

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

Aufrufe:

```
ggt 25 15 ->> 5        ggt 48 60 ->> 12
ggt 28 60 ->> 4        ggt 60 40 ->> 20
```

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 rekursiver Funktionen.

Aufrufe:

```
isEven 6 ->> True          iEven 9 ->> False
isOdd 6  ->> False         isOdd 9 ->> True
```

Beispiele – Die ersten Zehn im Rückblick

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

Beispiele – Die ersten Zehn im Rückblick (fgs.)

6. 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 funktionaler Programmierung (wie Prozeduren (Methoden) in prozeduraler (objektorientierter) Programmierung)

Funktionale Programme

- werten Ausdrücke aus. Das Resultat dieser Auswertung ist ein Wert von einem bestimmten Typ. Dieser Wert kann elementar oder funktional sein; er ist die Bedeutung, die Semantik dieses Ausdrucks.

Beispiel 1: Auswertung von Ausdrücken (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
```

Die einzelnen Vereinfachungs-, Rechenschritte werden wir später

- Simplifikationen

nennen.

Beispiel 1: Auswertung von Ausdrücken (2)

Dabei sind verschiedene Auswertungs-, Simplifizierungsreihenfolgen möglich:

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

    (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
```

Beispiel 2: Auswertung von Ausdrücken

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])
->> (1,2) : zip [3,5] [4,6,8,10]
->> (1,2) : zip (3:[5]) (4:[6,8,10])
->> (1,2) : ((3,4) : zip [5] [6,8,10])
->> (1,2) : ((3,4) : zip (5:[]) (6:[8,10]))
->> (1,2) : ((3,4) : ((5,6) : zip [] [8,10]))
->> (1,2) : ((3,4) : ((5,6) : []))
->> (1,2) : ((3,4) : [(5,6)])
->> (1,2) : [(3,4),(5,6)]
->> [(1,2),(3,4),(5,6)]
```

Beispiel 3: Auswertung von Ausdrücken (1)

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

Beispiel 3: Auswertung von Ausdrücken (2)

Eine 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
```

Beispiel 3: Auswertung von Ausdrücken (3)

Eine andere 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))))
```

Beispiel 3: Auswertung von Ausdrücken (4)

```
(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
```

In der Folge werden wir die mit

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

bezeichnen.

Die beiden Auswertungsreihenfolgen werden wir als

- applikative (unverzügliche) (1. Auswertungsf., z.B. in ML)
- normale (verzögerte) (2. Auswertungsf., z.B. in Haskell)

Auswertung voneinander abgrenzen.

"Finden" einer rekursiven Formulierung (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” einer rekursiven Formulierung (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” einer rekursiven Formulierung (3)

Wir erkennen:

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

Wir führen Sonder- und Regelfall zusammen und erhalten:

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

“Finden” einer rekursiven Formulierung (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” einer rekursiven Formulierung (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” einer rekursiven Formulierung (6)

Wir erkennen:

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

Wir führen Sonder- und Regelfall zusammen und erhalten:

```
natSum n = if n == 0 then 0
                     else (natSum (n-1)) + n
```

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
```

Haskell-Programme

...gibt es in zwei (notationellen) 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

FirstScript.hs: Gewöhnliches Haskell-Skript

```
{- +++ FirstScript.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))
```

FirstLitScript.lhs: Literates Haskell-Skript

```
+++ FirstLitScript.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
```

Wie in anderen Programmiersprachen

- haben Schlüsselwörter eine besondere Bedeutung und dürfen nicht als Identifikatoren für Funktionen oder Funktionsparameter verwendet werden.

Tipp

- Die Definition einiger der in diesem Kapitel beispielhaft betrachteten Rechenvorschriften und vieler weiterer allgemein nützlicher Rechenvorschriften findet sich in der Bibliotheksdatei
  - Prelude.hs
- Diese Bibliotheksdatei
  - wird automatisch mit jedem Haskell-Programm geladen, so dass die darin definierten Funktionen im Haskell-Programm benutzt werden können
  - ist quelloffen
- Nachschlagen und lesen in der Datei Prelude.hs ist daher eine gute und einfache Möglichkeit, sich mit der Syntax von Haskell vertraut zu machen und ein Gefühl für den Stil funktionaler Programmierung zu entwickeln.

Kapitel 1.2

Funktionale Programmierung: Warum? Warum mit Haskell?

Funktionale Programmierung: Warum?

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

John W. Backus, 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.

Funktionale Programmierung: Warum? (fgs.)

Es gibt einen bunten Strauß an *Programmierparadigmen*, z.B.:

- *imperativ*
  - prozedural (Pascal, Modula, C,...)
  - objektorientiert (Smalltalk, Java, C++, Eiffel,...)
  - parallel und verteilt (Ada, Mesa,...)
- *deklarativ*
  - funktional (Lisp, ML, Miranda, Haskell, Gofer,...)
  - logisch (Prolog, Datalog, Gödel,...)
  - Randbedingung (constraint) (Oz, Curry, Bertrand,...)
- *graphisch*
  - Graphische Programmiersprachen (Forms/3, FAR,...)
- Mischformen
  - Funktional/logisch (Curry, POPLOG, TOY, Mercury,...),
  - Funktional/objektorientiert (Haskell++, O'Haskell, OCaml,...)

Ein Vergleich - prozedural vs. funktional

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

Prozedural: Typischer Lösungsablauf in zwei Schritten:

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

Zentral:

- Der zweite Schritt erfordert zwingend, den Speicher explizit anzusprechen und zu verwalten (Allokation, Manipulation, Deallokation).

Ein einfaches Beispiel zur Illustration

Aufgabe:

- *"Bestimme in einem ganzzahligen Feld die Werte aller Komponenten mit einem Wert von höchstens* 10.*"*

Eine typische prozedurale Lösung, hier in Pascal:

```
VAR a, b: ARRAY [1..maxLength] OF integer;
...
j := 1;
FOR i:=1 TO maxLength DO
    IF a[i] <= 10 THEN
       BEGIN b[j] := a[i]; j := j+1 END;
```

Mögliches Problem, besonders bei sehr großen Anwendungen:

- Unzweckmäßiges Abstraktionsniveau $\rightsquigarrow$ Softwarekrise!

Beiträge zur Überwindung der Softwarekrise

Ähnlich wie objektorientierte Programmierung verspricht deklarative, speziell funktionale Programmierung

- dem Programmierer ein angemesseneres Abstraktionsniveau zur Modellierung und Lösung von Problemen zu bieten
- auf diese Weise einen Beitrag zu leisten
  - zur Überwindung der vielzitierten Softwarekrise
  - hin zu einer ingenieurmäßigen Software-Entwicklung ("in time, in functionality, in budget")

Zum Vergleich

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

```
a :: [Integer]
b :: [Integer]
...
b = [n | n<-a, n<=10]
```

Zentral:

- Keine Speichermanipulation, -verwaltung erforderlich.

Setze in Beziehung

- die funktionale Lösung `[n | n<-a, n<=10]` mit dem Anspruch
  - *"...etwas von der Eleganz der Mathematik in die Programmierung zu bringen!"*

    $\{n \mid n \in a \land n \leq 10\}$

Essenz funktionaler Programmierung

...und allgemeiner deklarativer Programmierung:

- *Statt des "wie" das "was" in den Vordergrund der Programmierung zu stellen!*

Ein wichtiges programmiersprachliches Hilfsmittel hierzu:

- Automatische Listengenerierung mittels Listenkomprehension (engl. list comprehension)

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

  $\leadsto$ typisch und kennzeichnend für funktionale Sprachen!

Noch nicht überzeugt?

Betrachte eine komplexere Aufgabe, Sortieren.

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung: 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) dem Element l sind.
- *Herrsche:* Sortiere L_1 und L_2 mithilfe des Quicksort-Verfahrens (d.h. mittels rekursiver Aufrufe von Quicksort).
- *Bestimme Gesamtlösung durch Zusammenführen der Teillösungen:* Hier trivial (konkateniere die sortierten Teillisten zur sortierten Gesamtliste).

Quicksort

...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]`.

Zum Vergleich

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

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

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)

Imperative Programmierung:

- Unterscheidung von Ausdrücken und Anweisungen.
- Ausdrücke liefern Werte. Anweisungen bewirken Zustandsänderungen (Seiteneffekte).
- Programmausführung meint Ausführung von Anweisungen; dabei müssen auch Ausdrücke ausgewertet werden.
- Kontrollflussspezifikation mittels spezieller Anweisungen (Fallunterscheidung, Schleifen, etc.)
- Variablen sind Verweise auf Speicherplätze. Ihre Werte können im Verlauf der Programmausführung geändert werden.

Imperative/Funktionale Programmierung (2)

Funktionale Programmierung:

- Keine Anweisungen, ausschließlich Ausdrücke.
- Ausdrücke liefern Werte. Zustandsänderungen (und damit Seiteneffekte) gibt es nicht.
- Programmausführung meint Auswertung eines 'Programmausdrucks'. Dieser beschreibt bzw. ist das Ergebnis des Programms.
- Keine Kontrollflussspezifikation; allein Datenabhängigkeiten regeln 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 somit 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: enorme Fortschritte sind gemacht (Performanz oft vergleichbar mit entsprechenden C-Implementierungen); Korrektheit zudem vorrangig gegenüber Geschwindigkeit; einfache(re) Parallelisierbarkeit fkt. Programme.
- Gelegentlich 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: 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`

Fkt. Programmierung: Warum mit Haskell?

Ein bunter Strauß funktionaler (Programmier-) sprachen, 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 McQueen)
- 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 – und zugleich eine gute Lehrsprache; auch dank Hugs!

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)

Kapitel 1.3

Nützliche Werkzeuge: Hugs, GHC, Hoogle und Hayoo, Leksah

Überblick

Beispielhaft 4 nützliche Werkzeuge für die funktionale Programmierung in Haskell:

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

1) Hugs

...ein populärer Haskell-Interpretierer:

- Hugs

Hugs im Netz:

- `www.haskell.org/hugs`

Hugs-Aufruf ohne Skript

Aufruf von Hugs ohne Skript:

```
hugs
```

Anschließend 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 Skript

Aufruf von Hugs mit Skript, z.B. mit `FirstScript.hs`:

```
hugs FirstScript.hs
```

Hugs-Aufruf allgemein: `hugs <filename>`

Bei Hugs-Aufruf mit Skript stehen zusätzlich auch alle im geladenen Skript deklarierten Funktionen zur Auswertung von Ausdrücken 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 (und ein eventuell vorher geladenes Skript zu ersetzen):

```
Main>:l SecondScript.lhs
```

Hugs – Wichtige 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 u. 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

Hugs – Fehlermeldungen u. Warnungen (fgs.)

Mehr zu Fehlermeldungen siehe z.B.:

```
www.cs.kent.ac.uk/
  people/staff/sjt/craft2e/errors.html
```

Professionell und praxisgerecht

- Haskell stellt umfangreiche Bibliotheken mit vielen vordefinierten Funktionen zur Verfügung.
- Die Standardbibliothek `Prelude.hs` wird automatisch beim Start von Hugs geladen. Sie stellt eine Vielzahl von Funktionen bereit, z.B. zum
  - Umkehren von Zeichenreichen, genereller von Listen (`reverse`)
  - Verschmelzen von Listen (`zip`)
  - Aufsummieren von Elementen einer Liste (`sum`)
  - ...

Namenskonflikte und ihre Vermeidung

...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 12 im Zusammenhang mit dem Modulkonzept von Haskell).

2) GHC

...ein populärer Haskell-Compiler:

- Glasgow Haskell Compiler (GHC)

...sowie ein von GHC abgeleiteter Interpretierer:

- GHCi (GHC interactive)

GHC (und GHCi) im Netz:

- `hackage.haskell.org/platform`

3) Hoogle und Hayoo

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

- Hoogle
- Hayoo

Hoogle und Hayoo unterstützen die Suche nach

- Funktionsnamen
- Modulnamen
- Funktionssignaturen

Hoogle und Hayoo im Netz:

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

4) 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.
- ...

4) Leksah

Leksah im Netz:

- www.leksah.org

Bemerkung:

- 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)

- 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. (Chapter 1, What is Functional Programming?)
- 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)
- 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)

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

- 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)
- Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. tryhaskell.org.

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

- 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.
- C.A.R. Hoare. *Quicksort*. The Computer Journal 5(1):10-15, 1962.

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

- 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, 2007. (Kapitel 1, Introduction; Kapitel 2, First Steps)
- Arjan van IJzendoorn, Daan Leijen, Bastiaan Heeren. *The Helium Compiler*. www.cs.uu.nl/helium.

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

- Mark P. Jones, Alastair Reid et al. (Hrsg.). *The Hugs98 User Manual*. `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 (7)

- 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 (8)

- 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 (9)

- 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 (10)

- Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3, Functional Programming Languages; Kapitel 3.1, Basic Concepts and Introductory Examples)

Kapitel 2

Grundlagen von Haskell

Kapitel 2.1

Elementare Datentypen

Überblick

Elementare Datentypen

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

Elementare Datentypen

...werden in der Folge nach nachstehendem Muster angegeben:

- Name des Typs
- Typische Konstanten des Typs
- Typische Operatoren (und Relatoren, so vorhanden)

Wahrheitswerte

Typ	`Bool`	Wahrheitswerte
Konstanten	`True :: Bool`	Symbol für 'wahr'
	`False :: Bool`	Symbol für 'falsch'
Operatoren	`(&&) :: Bool -> Bool -> Bool`	Logisches 'und'
	`(||) :: Bool -> Bool -> Bool`	Logisches 'oder'
	`not :: Bool -> Bool`	Logische Negation

Ganze Zahlen

Typ	`Int`	Ganze Zahlen (endlicher Ausschnitt)
Konstanten	`0 :: Int`	Symbol für '0'
	`-42 :: Int`	Symbol für '-42'
	`2147483647 :: Int`	Wert für 'maxInt'
	...	
Operatoren	`(+) :: Int -> Int -> Int`	Addition
	`(*) :: Int -> Int -> Int`	Multiplikation
	`(^) :: Int -> Int -> Int`	Exponentiation
	`(-) :: Int -> Int -> Int`	Subtraktion (Infix)
	`- :: Int -> Int`	Vorzeichenwechsel (Prefix)
	`div :: Int -> Int -> Int`	Division
	`mod :: Int -> Int -> Int`	Divisionsrest
	`abs :: Int -> Int`	Absolutbetrag
	`negate :: Int -> Int`	Vorzeichenwechsel

Ganze Zahlen (fgs.)

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

...die Relatoren == und /= sind auf Werte aller Elementar- und vieler weiterer Typen anwendbar, beispielsweise auch auf Wahrheitswerte (Stichwort: *Überladen* (engl. *Overloading*))!

...mehr zu Überladung in Kapitel 8.

Ganze Zahlen (nicht beschränkt)

Typ	`Integer`	Ganze Zahlen
Konstanten	`0   :: Integer`	Symbol für '0'
	`-42 :: Integer`	Symbol für '-42'
	`21474836473853883234 :: Integer`	'Große' Zahl
	...	
Operatoren	...	

...wie `Int`, jedoch ohne "*a priori*"-Beschränkung für eine maximal darstellbare Zahl.

Gleitkommazahlen

Typ	`Float`	Gleitkommazahlen (endl. Ausschnitt)
Konstanten	`0.123     :: Float`	Symbol für '0,123'
	`-47.11    :: Float`	Symbol für '-47,11'
	`123.6e-2  :: Float`	$123{,}6 \times 10^{-2}$
	...	
Operatoren	`(+) :: Float -> Float -> Float`	Addition
	`(*) :: Float -> Float -> Float`	Multiplikation
	...	
	`sqrt :: Float -> Float`	(pos.) Quadratwurzel
	`sin  :: Float -> Float`	sinus
	...	
Relatoren	`(==) :: Float -> Float -> Bool`	gleich
	`(/=) :: Float -> Float -> Bool`	ungleich
	...	

Gleitkommazahlen (fgs.)

Typ	`Double`	Gleitkommazahlen (endl. Ausschnitt)
	...	

Wie `Float`, jedoch mit doppelter Genauigkeit:

- `Float`: 32 bit
- `Double`: 64 bit

Zeichen

Typ	`Char`	Zeichen (Literal)
Konstanten	`‘a‘ :: Char`	Symbol für ‘a’
	...	
	`‘Z‘ :: Char`	Symbol für ‘Z’
	`‘\t‘ :: Char`	Tabulator
	`‘\n‘ :: Char`	Neue Zeile
	`‘\\‘ :: Char`	Symbol für ‘backslash’
	`‘\‘‘ :: Char`	Hochkomma
	`‘\"‘ :: Char`	Anführungszeichen
Operatoren	`ord :: Char -> Int`	Konversionsfunktion
	`chr :: Int -> Char`	Konversionsfunktion

Kapitel 2.2

Tupel und Listen

Überblick

- Tupel
  - Spezialfall: Paare
- Listen
  - Spezialfall: Zeichenreihen

Tupel und Listen

- Tupel
  - fassen eine vorbestimmte Zahl von Werten möglicherweise verschiedener Typen zusammen.
- Listen
  - fassen eine nicht vorbestimmte Zahl von Werten gleichen Typs zusammen.

Tupel

...fassen eine vorbestimmte Zahl von Werten möglicherweise verschiedener Typen zusammen.

$\rightsquigarrow$ Tupel sind heterogen!

Beispiele:

- Modellierung von Studentendaten:

```
("Max Muster","e123456@stud.tuwien.ac.at",534) ::
                                   (String, String, Int)
```

- Modellierung von Buchhandelsdaten:

```
("Simon Thompson","Haskell",3,2011,True) ::
                        (String, String, Int, Int, Bool)
```

Tupel (fgs.)

- Allgemeines Muster

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

  wobei `v1,...,vk` Bezeichnungen von Werten und `T1,...,Tk` Bezeichnungen von Typen sind mit

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

  Lies: `vi` ist vom Typ `Ti`

- Standardkonstruktor (runde Klammern)

```
(. , . , ... , . )
```

Tupel (fgs.)

Spezialfall: Paare ("Zweitupel")

- Beispiele

```
type Point = (Float, Float)

(0.0,0.0) :: Point
(3.14,17.4) :: Point
```

- Standardselektoren (für Paare)

```
fst (x,y) = x
snd (x,y) = y
```

- Anwendung der Standardselektoren

```
fst (1.0,2.0)   ->> 1.0
snd (3.14,17.4) ->> 17.4
```

Typsynonyme

...sind nützlich:

```
type Name  = String
type Email = String
type SKZ   = Int

type Student = (Name, Email, SKZ)
```

...erhöhen die Lesbarkeit und Transparenz in Programmen.

Wichtig: Typsynonyme definieren *keine* neuen Typen, sondern einen Namen für einen schon existierenden Typ (mehr dazu in Kapitel 6).

Typsynonyme (fgs.)

Typsynonyme für Buchhandelsdaten

```
type Autor = String
type Titel = String
type Auflage = Int
type Erscheinungsjahr = Int
type Lieferbar = Bool

type Buch = (Autor,Titel,Auflage,
             Erscheinungsjahr,Lieferbar)
```

Selektorfunktionen

Selbstdefinierte Selektorfunktionen

```
type Student = (Name, Email, SKZ)

name  :: Student -> Name
email :: Student -> Email
studKennZahl :: Student -> SKZ

name (n,e,k)         = n
email (n,e,k)        = e
studKennZahl (n,e,k) = k
```

...mittels Musterpassung (engl. pattern matching) (mehr dazu insbesondere in Kapitel 11).

Selektorfunktionen (fgs.)

Selektorfunktionen für Buchhandelsdaten

```
type Buch = (Autor,Titel,Auflage,
             Erscheinungsjahr,Lieferbar)

autor :: Buch -> Autor
titel :: Buch -> Titel
auflage :: Buch -> Auflage
erscheinungsjahr :: Buch -> Erscheinungsjahr
lieferbar :: Buch -> Lieferbar

autor (a,t,e,j,l)            = a
kurzTitel (a,t,e,j,l)        = t
auflage (a,t,e,j,l)          = e
erscheinungsJahr (a,t,e,j,l) = j
ausgeliehen (a,t,e,j,l)      = l
```

Listen

...fassen eine nicht vorbestimmte Zahl von Werten gleichen Typs zusammen.

$\leadsto$ Listen sind homogen!

Einfache Beispiele:

- Listen ganzer Zahlen
  `[2,5,12,42] :: [Int]`
- Listen von Wahrheitswerten
  `[True,False,True] :: [Bool]`
- Listen von Gleitkommazahlen
  `[3.14,5.0,12.21] :: [Float]`
- Leere Liste
  `[]`
- ...

Listen

Beispiele komplexerer Listen:

- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [[Int]]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3)] :: [(Float,Float)]`
- ...
- Listen von Funktionen
  `[fac, abs, negate] :: [Integer -> Integer]`

Vordefinierte Funktionen auf Listen

Die Funktion `length` mit einigen Aufrufen:

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

length [1, 2, 3]     ->> 3
length ['a','b','c'] ->> 3
length [[1],[2],[3]] ->> 3
```

Die Funktionen `head` und `tail` mit einigen Aufrufen:

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

head [[1],[2],[3]] ->> [1]
tail [[1],[2],[3]] ->> [[2],[3]]
```

Automatische Listengenerierung

- Listenkomprehension
  `list = [1,2,3,4,5,6,7,8,9]`
  `[3*n|n<-list]` kurz für `[3,6,9,12,15,18,21,24,27]`

  ⇝ Listenkomprehension ist ein sehr ausdruckskräftiges und elegantes Sprachkonstrukt, das eine automatische Generierung von Listen erlaubt!

- Spezialfälle (für Listen über geordneten Typen)
  - `[2..13]` kurz für `[2,3,4,5,6,7,8,9,10,11,12,13]`
  - `[2,5..22]` kurz für `[2,5,8,11,14,17,20]`
  - `[11,9..2]` kurz für `[11,9,7,5,3]`
  - `['a','d'..'j']` kurz für `['a','d','g','j']`
  - `[0.0,0.3..1.0]` kurz für `[0.0,0.3,0.6,0.9]`

Zeichenreihen: In Haskell spezielle Listen

Zeichenreihen sind in Haskell als Listen von Zeichen realisiert:

Typ	`String`	Zeichenreihen
Deklaration	`type String = [Char]`	Typsynonym (als Liste von Zeichen)
Konstanten	`"Haskell" :: String`	Zeichenreihe "Haskell"
	`""`	Leere Zeichenreihe
	...	
Operatoren	`(++) :: String -> String -> String`	Konkatenation
Relatoren	`(==) :: String -> String -> Bool`	gleich
	`(/=) :: String -> String -> Bool`	ungleich

Zeichenreihen (fgs.)

Beispiele:

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

Es gilt:

```
[1,2,3] == 1:2:3:[] == (1:(2:(3:[])))
```

Kapitel 2.3

Funktionen

Funktionen in Haskell

...am Beispiel der Fakultätsfunktion.

Aus der Mathematik:

$$! : IN \to IN$$

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

...*eine* mögliche Realisierung in Haskell:

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

Beachte: Haskell stellt eine Reihe oft knapperer und eleganterer notationeller Varianten zur Verfügung!

Fkt. in Haskell: Notationelle Varianten (1)

...am Beispiel der Fakultätsfunktion.

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

(1) In Form *"bedingter Gleichungen"*

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

Hinweis: Diese Variante ist "häufigst" benutzte Form!

Fkt. in Haskell: Notationelle Varianten (2)

(2) *Gleichungsorientiert*

$\text{fac n} = \text{if n} == 0 \text{ then } 1 \text{ else } (\text{n} * \text{fac } (\text{n} - 1))$

(3) *λ-artig* (argumentlos)

$\text{fac} = \backslash\text{n} \text{ -> } (\text{if n} == 0 \text{ then } 1 \text{ else } (\text{n} * \text{fac } (\text{n} - 1)))$

- Reminiszenz an den der funktionalen Programmierung zugrundeliegenden λ-Kalkül ($\lambda\, x\, y.\ (x + y)$)
- In Haskell: $\backslash\, x\ y \rightarrow\ x + y$ sog. anonyme Funktion

  Praktisch, wenn der Name keine Rolle spielt und man sich deshalb bei Verwendung von anonymen Funktionen keinen zu überlegen braucht.

Fkt. in Haskell: Notationelle Varianten (3)

(4) Mittels *lokaler Deklarationen*

- (4a) *where*-Konstrukt
- (4b) *let*-Konstrukt

...am Beispiel der Funktion `quickSort`.

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

Fkt. in Haskell: Notationelle Varianten (4)

(4a) *where*-Konstrukt

```
quickSort []     = []
quickSort (x:xs) = quickSort allSmaller ++
                   [x] ++ quickSort allLarger
                   where
                    allSmaller = [ y | y<-xs, y<=x ]
                    allLarger  = [ z | z<-xs, z>x ]
```

(4b) *let*-Konstrukt

```
quickSort []     = []
quickSort (x:xs) = let
                    allSmaller = [ y | y<-xs, y<=x ]
                    allLarger  = [ z | z<-xs, z>x ]
                   in (quickSort allSmaller ++
                       [x] ++ quickSort allLarger)
```

Fkt. in Haskell: Notationelle Varianten (5)

(5) Mittels *lokaler Deklarationen*

- (5a) *where*-Konstrukt
- (5b) *let*-Konstrukt

in einer Zeile.

...am Beispiel der Funktion `kAV` zur Berechnung von Oberfläche (A) und Volumen (V) einer Kugel (k) mit Radius (r).

Für die Berechnung von A und V von k mit Radius r gilt:

- $A = 4\,\pi\,r^2$
- $V = \frac{4}{3}\,\pi\,r^3$

Fkt. in Haskell: Notationelle Varianten (6)

```
type Area   = Float
type Volume = Float
type Radius = Float
kAV :: Radius -> (Area,Volume)
```

In einer Zeile

(5a) Mittels "where" und ";"

```
kAV r =
  (4*pi*square r, (4/3)*pi*cubic r)
   where
    pi = 3.14; cubic x = x*square x; square x = x*x
```

(5b) Mittels "let" und ";"

```
kAV r =
  let pi = 3.14; cubic x = x*square x; square x = x*x
  in (4*pi*square r, (4/3)*pi*cubic r)
```

Fkt. in Haskell: Notationelle Varianten (7)

Spezialfall: *Binäre* (*zweistellige*) Funktionen

```
biMax :: Int -> Int -> Int
biMax p q
  | p >= q    = p
  | otherwise = q

triMax :: Int -> Int -> Int -> Int
triMax p q r
  | (biMax p q == p) && (p `biMax` r == p) = p
  | ...
  | otherwise                              = r
```

Beachte: `biMax` ist in `triMax` als *Präfix-* (`biMax p q`) und als *Infixoperator* (`` p `biMax` r ``) verwandt.

Fkt. in Haskell: Notationelle Varianten (8)

Musterbasiert

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

Beachte:

- Die Klammerung im Ausdruck `fib (n-2) + fib (n-1)` ist erforderlich:

  `fib n-2 + fib n-1` steht abkürzend für
  `((((fib n) - 2) + (fib n)) - 1)`

  `fib (n-2) + fib (n-1)` steht entsprechend für
  `((fib (n-2)) + (fib (n-1)))`

  ⇝ Linksassoziative Klammerung für Ausdrücke
  (vgl. Klammereinsparungsregeln (Kapitel 2.5))

Fkt. in Haskell: Notationelle Varianten (10)

Vergleiche die musterbasierte Definition von fib

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

mit folgenden gleichungsbasierten Varianten auf “Einfachheit”:

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

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

Fkt. in Haskell: Notationelle Varianten (10)

Musterbasiert (fgs.)

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

Fkt. in Haskell: Notationelle Varianten (11)

Mittels *case*-Ausdrucks

```
capVowels :: Char -> Char
capVowels letter
  = case letter of
    ‘a‘     -> ‘A‘
    ‘e‘     -> ‘E‘
    ‘i‘     -> ‘I‘
    ‘o‘     -> ‘O‘
    ‘u‘     -> ‘U‘
    letter  -> letter
```

```
decapVowels :: Char -> Char
decapVowels letter
  = case letter of
    ‘A‘        -> ‘a‘
    ‘E‘        -> ‘e‘
    ‘I‘        -> ‘i‘
    ‘O‘        -> ‘o‘
    ‘U‘        -> ‘u‘
    otherwise  -> letter
```

Fkt. in Haskell: Notationelle Varianten (12)

Mittels *Muster* und *"wild cards"*

```
myAdd :: Int -> Int -> Int -> Int
myAdd n 0 0 = n
myAdd 0 n 0 = n
myAdd 0 0 n = n
myAdd m n p = m+n+p
```

```
myMult :: Int -> Int -> Int -> Int
myMult 0 _ _ = 0
myMult _ 0 _ = 0
myMult _ _ 0 = 0
myMult m n p = m*n*p
```

Muster

...sind (u.a.):

- Werte (z.B. 0, ‘c‘, True)
  ...ein Argument “passt” auf das Muster, wenn es vom entsprechenden Wert ist.
- Variablen (z.B. n)
  ...jedes Argument passt.
- Wild card “_”
  ...jedes Argument passt (Benutzung von “_” ist sinnvoll für solche Argumente, die nicht zum Ergebnis beitragen).
- ...

$\rightsquigarrow$ weitere Muster und mehr über musterbasierte Funktionsdefinitionen im Lauf der Vorlesung, insbesondere in Kapitel 11.

Zum Abschluss: Ein "Anti"-Beispiel

Nicht alles, was syntaktisch möglich ist, ist semantisch sinn- und bedeutungsvoll.

Betrachte folgende "Funktions"-Definition:

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

Zum Abschluss: Ein "Anti"-Beispiel (fgs.)

Beachte: Die Funktion `f` ist nur an der Stelle 1 definiert!

Wir erhalten: `f 1 ->> 2`

Aber:

```
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))) ->>...

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

Zum Abschluss: Ein “Anti”-Beispiel (fgs.)

- `f` wird nicht als sinnvolle Funktionsdefinition angesehen (auch wenn `f` formal eine partiell definierte Funktion festlegt).

Zum Vergleich

Die Funktionen g und h legen in sinnvoller (wenn auch für h in unüblicher) Weise partielle Funktionen fest:

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

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

Zum Vergleich (fgs.)

Wir erhalten:

```
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
```

Aber:

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

Zum Vergleich (fgs.)

Wir erhalten:

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

Aber:

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

Beobachtung

Die Funktionen g und h legen partielle Funktionen fest, und zwar:

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

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

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

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

Beobachtung (fgs.)

Zur Deutlichkeit wäre somit besser:

```
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"
```

Sowie:

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

Kapitel 2.4

Programmlayout und Abseitsregel

Layout-Konventionen für Haskell-Programme

Für die meisten Programmiersprachen gilt:

- Das Layout eines Programms hat Einfluss
  - auf seine Lesbarkeit, Verständlichkeit, Wartbarkeit
  - aber nicht auf seine Bedeutung

Für Haskell gilt das nicht!

Für Haskell gilt:

- Das Layout eines Programms trägt Bedeutung!
- Für Haskell ist für diesen Aspekt des Sprachentwurfs eine grundsätzlich andere Entwurfsentscheidung getroffen worden als z.B. für Java, Pascal, C u.a.
- Dies ist Reminiszenz an Cobol, Fortran. Layoutabhängigkeit ist aber auch zu finden in anderen modernen Sprachen, z.B. occam.

Abseitsregel (engl. offside rule)

Layout-abhängige Syntax als notationelle Besonderheit in Haskell.

"Abseits"-Regel

- Erstes Zeichen einer Deklaration (bzw. nach `let`, `where`):
  $\rightsquigarrow$ *Startspalte neuer "Box" (Bindungsbereichs) wird festgelegt*
- Neue Zeile
  - gegenüber der aktuellen Box nach rechts eingerückt:
    $\rightsquigarrow$ *aktuelle Zeile wird fortgesetzt*
  - genau am linken Rand der aktuellen Box:
    $\rightsquigarrow$ *neue Deklaration wird eingeleitet*
  - weiter links als die aktuelle Box:
    $\rightsquigarrow$ *aktuelle Box wird beendet ("Abseitssituation")*

Ein Beispiel zur Abseitsregel

Unsere Funktion `kAV` zur Berechnung von Oberfläche und Volumen einer Kugel mit Radius `r`:

```
kAV r =
  (4*pi*square r, (4/3)*pi*cubic r)
  where
  pi = 3.14
  cubic x = x *
      square x

square x = x * x
```

...ist kein schönes, aber (Haskell-) korrektes Layout.

Das Layout genügt der Abseitsregel von Haskell und damit den Layout-Anforderungen.

Ein Beispiel zur Abseitsregel (fgs.)

Graphische Veranschaulichung der Abseitsregel:

```
------------------------------------------------
|
kAV r =
|  (4*pi*square r, (4/3)*pi*cubic r)
   ---------------------------------
|  |
|  where
|  pi = 3.14
|  cubic x = x *
|  |   square x
|  --------------------->
------------------------------------>
------------------------------------------------
square x = x * x
|
\/
```

Layout-Konventionen

Es ist bewährt, folgende Layout-Konvention einzuhalten:

```
funName f1 f2... fn
  | g1    = e1
  | g2    = e2
  ...
  | gk    = ek

funName f1 f2... fn
  | diesIsteineGanz
    BesondersLange
    Bedingung
       = diesIstEinEbenso
         BesondersLangerAusdruck
  | g2         = e2
  ...
  | otherwise = ek
```

Angemessene Notationswahl

...nach Zweckmäßigkeitserwägungen.

- Auswahlkriterium:
  Welche Variante lässt sich am einfachsten verstehen?

Zur Illustration:

- Vergleiche folgende 3 Implementierungsvarianten der Rechenvorschrift

```
triMax :: Int -> Int -> Int -> Int
```

zur Bestimmung des Maximums dreier ganzer Zahlen.

Angemessene Notationswahl (fgs.)

```
triMax :: Int -> Int -> Int -> Int

a) triMax = \p q r ->
     if p>=q then (if p>=r then p
                           else r)
             else (if q>=r then q
                           else r)
b) triMax p q r =
     if (p>=q) && (p>=r) then p
                         else
     if (q>=p) && (q>=r) then q
                         else r
c) triMax p q r
     | (p>=q) && (p>=r)  = p
     | (q>=p) && (q>=r)  = q
     | (r>=p) && (r>=q)  = r
```

Resümee und Fazit

Hilfreich ist folgende Richtschnur von C.A.R. "Tony" Hoare:

Programme können grundsätzlich auf zwei Arten geschrieben werden:

- So einfach, dass sie offensichtlich keinen Fehler enthalten
- So kompliziert, dass sie keinen offensichtlichen Fehler enthalten

Die Auswahl einer zweckmäßigen Notation trägt dazu bei!

Kapitel 2.5

Funktionssignaturen, -terme und -stelligkeiten

Überblick

In der Folge beschäftigen wir uns mit

- (Funktions-) Signaturen
- (Funktions-) Termen
- (Funktions-) Stelligkeiten

in Haskell.

Das Wichtigste

...in Kürze vorweg zusammengefasst:

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

in Haskell.

Durchgehendes Beispiel

Wir betrachten einen einfachen Editor `Edt` und eine Funktion `ers`, die in diesem Editor ein bestimmtes Vorkommen einer Zeichenreihe s durch eine andere Zeichenreihe t ersetzt.

In Haskell können `Edt` und `ers` wie folgt deklariert sein:

```
type Edt = String
type Vork = Integer
type Alt = String
type Neu = String

ers :: Edt -> Vork -> Alt -> Neu -> Edt
```

Abbildungsidee: Angewendet auf einen Editor e, eine ganze Zahl n, eine Zeichenreihe s und eine Zeichenreihe t ist das Resultat der Funktionsanwendung von `ers` ein Editor, in dem das n-te Vorkommen von s in e durch t ersetzt ist.

Funktionssignatur und Funktionsterm

- Funktionssignaturen (auch: syntaktische Funktionssignatur oder Signatur) geben den Typ einer Funktion an
- Funktionsterme sind aus Funktionsaufrufen aufgebaute Ausdrücke

Beispiele:

- Funktionssignatur

```
ers :: Edt -> Vork -> Alt -> Neu -> Edt
```

- Funktionsterm

```
ers "dies ist text" 1 "text" "neuer text"
```

Klammereinsparungsregeln

...für Funktionssignaturen und Funktionsterme.

Folgende Klammereinsparungsregeln gelten:

- Für Funktionssignaturen Rechtsassoziativität

  ```
  ers :: Edt -> Vork -> Alt -> Neu -> Edt
  ```

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

  ```
  ers :: (Edt -> (Vork -> (Alt -> (Neu -> Edt))))
  ```

- Für Funktionsterme Linksassoziativität

  ```
  ers "dies ist text" 1 "text" "neuer text"
  ```

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

  ```
  ((((ers "dies ist text") 1) "text") "neuer text")
  ```

Klammereinsparungsregeln (fgs.)

Die Festlegung von

- Rechtsassoziativität für Signaturen
- Linksassoziativität für Funktionsterme

dient der Einsparung von

- Klammern in Signaturen und Funktionstermen (vgl. Punkt- vor Strichrechnung)

Die Festlegung ist so erfolgt, da man auf diese Weise

- in Signaturen und Funktionstermen oft vollkommen ohne Klammern auskommt.

Typen von Funktionen u. Funktionstermen (1)

- Die Funktion `ers` ist Wert vom Typ `(Edt -> (Vork -> (Alt -> (Neu -> Edt))))`, d.h.

```
ers :: (Edt -> (Vork -> (Alt -> (Neu -> Edt))))
```

- Der Funktionsterm `((((ers "dies ist text") 1) "text") "neuer text")` ist Wert vom Typ `Edt`, d.h.

```
((((ers "dies ist text") 1) "text") "neuer text")
                                               :: Edt
```

Typen von Funktionen u. Funktionstermen (2)

Nicht nur die Funktion `ers`, auch die Funktionsterme nach Argumentkonsumation sind Werte von einem Typ, bis auf den letzten von einem funktionalen Typ.

```
ers :: (Edt -> (Vork -> (Alt -> (Neu -> Edt))))
```

Im einzelnen:

- Die Funktion `ers` konsumiert ein Argument vom Typ `Edt` und der resultierende Funktionsterm ist selbst wieder eine Funktion, eine Funktion vom Typ `(Vork -> (Alt -> (Neu -> Edt)))`:

```
(ers "dies ist text") :: (Vork -> (Alt -> (Neu -> Edt)))
```

Typen von Funktionen u. Funktionstermen (3)

- Die Funktion `(ers "dies ist text")` konsumiert ein Argument vom Typ `Vork` und der resultierende Funktionsterm ist selbst wieder eine Funktion, eine Funktion vom Typ `(Alt -> (Neu -> Edt))`:

```
((ers "dies ist text") 1) :: (Alt -> (Neu -> Edt))
```

- Die Funktion `((ers "dies ist text") 1)` konsumiert ein Argument vom Typ `Alt` und der resultierende Funktionsterm ist selbst wieder eine Funktion, eine Funktion vom Typ `(Neu -> Edt)`:

```
(((ers "dies ist text") 1) "text") :: (Neu -> Edt)
```

Typen von Funktionen u. Funktionstermen (4)

- Die Funktion `(((ers "dies ist text") 1) "text")` konsumiert ein Argument vom Typ `Neu` und ist von einem elementaren Typ, dem Typ `Edt`:

```
((((ers "dies ist text") 1) "text") "neuer Text")
                                            :: Edt
```

Stelligkeit von Funktionen in Haskell

Das vorige Beispiel illustriert:

- Funktionen in Haskell sind einstellig
- Funktionen in Haskell
  - konsumieren ein Argument und liefern ein Resultat eines funktionalen oder elementaren Typs

Funktionspfeil vs. Kreuzprodukt

Zwei naheliegende Fragen im Zusammenhang mit der Funktion `ers`:

- Warum so viele Pfeile (->) in der Signatur von `ers`, warum so wenige Kreuze ($\times$)?
- Warum nicht eine Signaturzeile im Stile von "ers :: (Edt × Vork × Alt × Neu) -> Edt"

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

  ```
  ers :: (Edt,Vork,Alt,Neu) -> Edt
  ```

Die Antwort

- Beide Formen sind möglich und üblich; beide sind sinnvoll und berechtigt
- "Funktionspfeil" führt i.a. jedoch zu höherer (Anwendungs-) Flexibilität als "Kreuzprodukt"
  - "Funktionspfeil" ist daher in funktionaler Programmierung die häufiger verwendete Form

Zur Illustration ein kompakteres Beispiel:

- Binomialkoeffizienten

Funktionspfeil vs. Kreuzprodukt

...am Beispiel der Berechnung der Binomialkoeffizienten:

Vergleiche die Funktionspfeilform

```
binom1 :: Integer -> Integer -> Integer
binom1 n k
 | k==0 || n==k = 1
 | otherwise    = binom1 (n-1) (k-1) + binom1 (n-1) k
```

mit der Kreuzproduktform

```
binom2 :: (Integer,Integer) -> Integer
binom2 (n,k)
 | k==0 || n==k = 1
 | otherwise    = binom2 (n-1,k-1) + binom2 (n-1,k)
```

Funktionspfeil vs. Kreuzprodukt (fgs.)

Die höhere Anwendungsflexibilität der Funktionspfeilform zeigt sich in der Aufrufsituation:

- Der Funktionsterm `binom1 45` ist von funktionalem Typ:

  ```
  binom1 45 :: Integer -> Integer
  ```

- Der Funktionsterm `binom1 45` (zur Deutlichkeit klammern wir: `(binom1 45)`) bezeichnet eine Funktion, die ganze Zahlen in sich abbildet
- Präziser: Angewendet auf eine natürliche Zahl k liefert der Aufruf der Funktion `(binom1 45)` die Anzahl der Möglichkeiten, auf die man k Elemente aus einer 45-elementigen Grundgesamtheit herausgreifen kann (“k aus 45”)

Funktionspfeil vs. Kreuzprodukt (fgs.)

- Wir können den Funktionsterm `(binom1 45)`, der funktionalen Typ hat, deshalb auch benutzen, um eine neue Funktion zu definieren. Z.B. die Funktion aus45. Wir definieren sie in argumentfreier Weise:

```
aus45 :: Integer -> Integer
aus45 = binom1 45  -- arg.frei: arg45 ist nicht
                   -- von einem Arg. gefolgt
```

- Die Funktion `aus45` u. der Funktionsterm `(binom1 45)` sind "Synonyme"; sie bezeichnen dieselbe Funktion.
- Folgende Aufrufe sind (beispielsweise) jetzt möglich:

```
(binom1 45) 6 ->> 8.145.060
binom1 45 6   ->> 8.145.060  -- wg. Linksass.
aus45 6       ->> 8.145.060
```

```
Im Detail: aus45 6 ->> binom1 45 6 ->> 8.145.060
```

Funktionspfeil vs. Kreuzprodukt (fgs.)

- Auch die Funktion

  ```
  binom2 :: (Integer,Integer) -> Integer
  ```

  ist (im Haskell-Sinn) einstellig.

  Ihr eines Argument p ist von einem Paartyp, dem Paartyp `(Integer,Integer)`.

- Die Funktion binom2 erlaubt die Anwendungsflexibilität der Funktion binom1 allerdings nicht.

  binom2 konsumiert ihr eines Argument p vom Paartyp `(Integer,Integer)` und liefert ein Resultat vom elementaren Typ `Integer`; ein funktionales Zwischenresultat entsteht (anders als bei binom1) nicht.

Funktionspfeil vs. Kreuzprodukt (fgs.)

- Der Aufruf von binom2 mit einem Wertepaar als Argument liefert sofort einen Wert elementaren Typs, keine Funktion

```
binom2 (45,6) ->> 8.145.060 :: Integer
```

- Eine nur "partielle" Versorgung mit Argumenten ist (anders als bei binom1) nicht möglich.

  Aufrufe der Art

  ```
  binom2 45
  ```

  sind syntaktisch inkorrekt und liefern eine Fehlermeldung.

- Insgesamt: Geringere (Anwendungs-) Flexibilität

Weitere Beispiele: Arithmetische Funktionen

Auch die arithmetischen Funktionen sind in Haskell curryfiziert (d.h. in der Funktionspfeilform) vordefiniert:

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

Nachstehend instantiiert für den Typ `Integer`:

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

Spezielle arithmetische Funktionen

Häufig sind folgende Funktionen benötigt/vordefiniert:

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

Spezielle arithmetische Funktionen (2)

Mögl. Standardimplementierungen (mittels Infixoperatoren):

- 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
```

- Verdoppeln

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

- 10er-Inkrement

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

Spezielle arithmetische Funktionen (3)

Mögl. Standardimplementierungen (mittels Präfixoperatoren):

- Inkrement

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

- Dekrement

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

- Halbieren

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

- Verdoppeln

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

- 10er-Inkrement

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

Spezielle arithmetische Funktionen (4)

Die curryfiz. spezifiz. arithm. Std.-Funktionen erlaubten auch:

- Inkrement

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

- Dekrement

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

- Halbieren

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

- Verdoppeln

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

- 10er-Inkrement

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

Spezielle arithmetische Funktionen (5)

Beachte:

- Die unterschiedliche Klammerung/Operatorverwendung bei `inc` und `dec` sowie bei `hlv` und `dbl`

Der Grund:

- Subtraktion und Division sind nicht kommutativ
- Infix- und Präfixbenutzung machen für nicht-kommutative Operatoren einen Bedeutungsunterschied
  - Infixbenutzung führt zu den Funktionen `inc` und `hlv`
  - Präfixbenutzung führt zu den Funktionen `einsMinus` und `zweiDurch`

Spezielle arithmetische Funktionen (6)

Im einzelnen für `dec` und `einsMinus`:

- Dekrement ("minus eins") ((-1) Infixoperator)

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

```
dec 5 ->> 4, dec 10 ->> 9, dec (-1) ->> -2
```

- "eins minus" ((-) Präfixoperator)

```
einsMinus :: Integer -> Integer
einsMinus = (-) 1    -- gleichwertig zu: (1-)
```

```
einsMinus 5 ->> -4, einsMinus 10 ->> -9,
einsMinus (-1) ->> 2
```

Spezielle arithmetische Funktionen (7)

Im einzelnen für `hlv` und `zweiDurch`:

- Halbieren ("durch zwei") (`‘div‘` Infix-Operator)

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

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

- "zwei durch" (`div` Präfixoperator)

```
zweiDurch :: Integer -> Integer
zweiDurch = (div 2)  -- gleichw. zu: (2 ‘div‘)
```

```
zweiDurch 5 ->> 0, zweiDurch 10 ->> 0,
zweiDurch 15 ->> 0
```

Spezielle arithmetische Funktionen (8)

Für `inc` ergibt sich wg. der Kommutativität der Addition kein Unterschied:

- Inkrement ((+1), Infix-Benutzung von (+))

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

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

- Inkrement ((+) Präfixoperator)

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

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

Spezielle arithmetische Funktionen (9)

Auch für `dbl` ergibt sich wg. der Kommutativität der Multiplikation kein Unterschied:

- Verdoppeln ((*2), Infix-Benutzung von (*))

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

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

- Verdoppeln ((*) Präfixoperator)

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

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

Anmerkungen zu Operatoren in Haskell

Operatoren in Haskell sind

- grundsätzlich Präfixoperatoren; das gilt insbesondere für alle selbstdeklarierten Operatoren (d.h. selbstdeklarierte Funktionen)

  Beispiele: `fac 5`, `binom1 45 6`, `triMax 2 5 3`,...

- in wenigen Fällen grundsätzlich Infixoperatoren; das gilt insbesondere für die arithmetischen Standardoperatoren

  Beispiele: `2+3`, `3*5`, `7-4`, `5^3`,...

Spezialfall: Binäre Operatoren in Haskell

Für binäre Operatoren gelten in Haskell erweiterte Möglichkeiten. Sowohl

- Infix- wie Präfixverwendung ist möglich!

Im Detail:

Sei `bop` binärer Operator in Haskell:

- Ist `bop` standardmäßig
  - präfix-angewendet, kann `bop` in der Form `‘bop‘` als Infixoperator verwendet werden

    Beispiel: `45 ‘binom1‘ 6`
    (statt standardmäßig `binom1 45 6`)
  - infix-angewendet, kann `bop` in der Form `(bop)` als Präfixoperator verwendet werden

    Beispiel: `(+) 2 3` (statt standardmäßig `2+3`)

Spezialfall: Binärop. in Operatorabschnitten

Partiell mit Operanden versorgte Binäroperatoren heißen im Haskell-Jargon

- 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<)` `zweiKleiner`, das Prädikat, das überprüft, ob sein Argument größer als 2 ist $(\lambda x.\, 2 < x)$
- `(<2)` `kleiner2`, das Prädikat, das überprüft, ob sein Argument kleiner als 2 ist $(\lambda x.\, x < 2)$
- `(2:)` `headApp`, die Funktion, die 2 an den Anfang einer typkompatiblen Liste setzt
- ...

Spezialfall: Binärop. in Operatorabschnitten

Weitere Operatorabschnittbeispiele:

- `(-1)` `dec`, die Funktion, die ihr Argument um 1 verringert $(\lambda x.\, x - 1)$
- `(1-)` `einsMinus`, die Funktion, die ihr Argument von 1 abzieht $(\lambda x.\, 1 - x)$
- ``(`div` 2)`` `hlv`, die Funktion, die ihr Argument ganzzahlig halbiert $(\lambda x.\, x \ \mathit{div}\ 2)$
- ``(2 `div`)`` `zweiDurch`, die Funktion, die 2 ganzzahlig durch ihr Argument teilt $(\lambda x.\, 2 \ \mathit{div}\ x)$
- `(div 2)` `zweiDurch`, s.o. $(\lambda x.\, 2 \ \mathit{div}\ x)$
- `div 2` `zweiDurch`, s.o. (wg. Linksass.); wg. fehlender Klammerung kein Operatorabschnitt, sondern normale Präfixoperatorverwendung.
- ...

Spezialfall: Binärop. in Operatorabschnitten

Operatorabschnitte können in Haskell

- auch mit selbstdefinierten binären Operatoren (d.h. Funktionen)

gebildet werden.

Beispiele:

- `(binom1 45)` `aus45`, die Funktion "k aus 45".
- `(45 ‘binom1‘)` `aus45`, s.o.
- `(‘binom1‘ 6)` `sechsAus`, die Funktion "6 aus n".
- ...

Beachte: Mit `binom2` können keine Operatorabschnitte gebildet werden.

Operatorabschnitte zur Funktionsdefinition

- "k aus 45"

```
aus45 :: Integer -> Integer
aus45 = binom1 45
```

```
aus45 :: Integer -> Integer
aus45 = ( 45 `binom1`)
```

- "6 aus n"

```
sechsAus :: Integer -> Integer
sechsAus = (`binom1` 6)
```

- Inkrement

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

- Halbieren

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

- ...

Funktionsstelligkeit: Mathematik vs. Haskell

Unterschiedliche Sichtw. in Mathematik und Programmierung

Mathematik: Eine Funktion der Form

$$\binom{\cdot}{\cdot} : IN \times IN \rightarrow IN$$

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

wird als zweistellig angesehen (die “Teile” werden betont).

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

Haskell: Eine Funktion der Form

```
binom2 :: (Integer,Integer) -> Integer
binom2 (n,k)
  | k==0 || n==k = 1
  | otherwise = binom2 (n-1,k-1) + binom2 (n-1,k)
```

wird als einstellig angesehen (das “Ganze” wird betont).

Funktionsstelligkeit in Haskell – Intuition

Musterverzicht in der Deklaration lässt die in Haskell verfolgte Intention deutlicher hervortreten:

- binom2 ohne Musterverwendung:

```
type IntegerPair = (Integer,Integer)
binom2 :: IntegerPair -> Integer -- 1 Argument!
binom2 p                         -- 1-stellig!
 | snd(p) == 0 || fst(p)==snd(p) = 1
 | otherwise = binom2 (fst(p)-1,snd(p)-1)
                       + binom2 (fst(p)-1,snd(p))
```

```
binom2 (45,6) ->> 8.145.060
```

...aber auch den Preis des Verzichts auf Musterverwendung:

- Abstützung auf Selektorfunktionen
- Verlust an Lesbarkeit und Transparenz

Nutzen von Musterverwendung

Zum Vergleich noch einmal die Deklaration unter Musterverwendung:

- binom2 mit Musterverwendung:

```
binom2 :: (Integer,Integer) -> Integer
binom2 (n,k)
 | k==0 || n==k = 1
 | otherwise = binom2 (n-1,k-1) + binom2 (n-1,k)
```

```
binom2 (45,6) ->> 8.145.060
```

Vorteile von Musterverwendung:

- Keine Abstützung auf Selektorfunktionen
- Gewinn an Lesbarkeit und Transparenz

Resümee

- Die musterlose Spezifikation von `binom2` macht die "Aufeinmalkonsumation" der Argumente besonders augenfällig.
- Der Vergleich der musterlosen und musterbehafteten Spezifikation von `binom2` zeigt den Vorteil von Musterverwendung:
  - Muster ermöglichen auf explizite Selektorfunktionen zu verzichten.
  - Implementierungen werden so kompakter und verständlicher.

Kapitel 2.6

Mehr Würze: Curry bitte!

Darf es etwas schärfer sein?

- Curry, bitte. Curryfizieren!

Jetzt wird's "hot"!

Curryfiziert

- steht für eine bestimmte Deklarationsweise von Funktionen. Decurryfiziert auch.

Maßgeblich

- ist dabei die Art der Konsumation der Argumente.

Erfolgt

- die Konsumation mehrerer Argumente durch Funktionen
  - einzeln Argument für Argument: curryfiziert
  - gebündelt als Tupel: decurryfiziert

Implizit

- liefert dies eine Klassifikation von Funktionen.

“Hot” vs. “Mild”

Beispiel:

- binom1 ist curryfiziert deklariert:

```
binom1 :: Integer -> Integer -> Integer
```

- binom2 ist decurryfiziert deklariert:

```
binom2 :: (Integer,Integer) -> Integer
```

“Hot” vs. “Mild” (fgs.)

Beispiel (fgs.):

- Curryfiziert deklariertes binom1:

```
binom1 :: Integer -> Integer -> Integer
binom1 n k
 | k==0 || n==k = 1
 | otherwise = binom1 (n-1) (k-1) + binom1 (n-1) k
```

```
(binom1 45) 6) ->> 8.145.060
```

- Decurryfiziert deklariertes binom2:

```
binom2 :: (Integer,Integer) -> Integer
binom2 (n,k)
 | k==0 || n==k = 1
 | otherwise = binom2 (n-1,k-1) + binom2 (n-1,k)
```

```
binom2 (45,6) ->> 8.145.060
```

Curry und uncurry: Zwei Funktionale als Mittler zwischen "hot" und "mild"

Informell:

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

Bemerkung: Die Bezeichnung erinnert an Haskell B. Curry; die (weit ältere) Idee geht auf Moses Schönfinkel aus der Mitte der 1920er-Jahre zurück.

Curry und uncurry: Zwei Funktionale als Mittler zwischen "hot" und "mild" (fgs.)

Zentral:

- die Funktionale (synonym: Funktionen höherer Ordnung) `curry` und `uncurry`

```
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
```

Die Funktionale curry und uncurry

Die Funktionale curry und uncurry bilden

- decurryfizierte Funktionen auf ihr curryfiziertes Gegenstück ab, d.h. für decurryfiziertes
  `f :: (a,b) -> c` ist

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

  curryfiziert.

- curryfizierte Funktionen auf ihr decurryfiziertes Gegenstück ab, d.h. für curryfiziertes
  `g :: a -> b -> c` ist

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

  decurryfiziert.

Anwendungen von curry und uncurry

Betrachte

```
binom1 :: (Integer -> Integer -> Integer)
binom2 :: ((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 binom2 :: (Integer -> Integer -> Integer)
uncurry binom1 :: ((Integer,Integer) -> Integer)
```

Somit sind folgende Aufrufe gültig:

```
curry binom2 45 6 ->> binom2 (45,6) ->> 8.145.060

uncurry binom1 (45,6) ->> binom1 45 6 ->> 8.145.060
```

Curry- oder decurryfiziert, “hot” oder “mild”?

...das ist hier die Frage.

Zum einen:

- Geschmackssache (sozusagen eine notationelle Spielerei)
  ...auch das, aber: die Verwendung curryfizierter Formen ist in der Praxis vorherrschend
  ⇝ `f x`, `f x y`, `f x y z`,... möglicherweise eleganter empfunden als `f x`, `f(x,y)`, `f(x,y,z)`,...

Zum anderen (und gewichtiger!):

- Sachargument
  ...(nur) Funktionen in curryfizierter Darstellung unterstützen partielle Auswertung
  ⇝ Funktionen liefern Funktionen als Ergebnis!

  Beispiel: `binom1 45 :: Integer -> Integer` ist eine einstellige Funktion auf den ganzen Zahlen; sie entspricht der Funktion `aus45`.

Mischformen

Neben den beiden Polen "hot" und "mild"

- "rein" curryfiziert (d.h. rein funktionspfeilorientiert)
  ```
  ers :: Edt -> Vork -> Alt -> Neu -> Edt
  ```
- "rein" decurryfiziert (d.h. rein kreuzproduktorientiert)
  ```
  ers :: (Edt,Vork,Alt,Neu) -> Edt
  ```

...sind auch Mischformen möglich und (zumeist) sinnvoll:

```
ers :: Edt -> Vork -> (Alt,Neu) -> Edt

ers :: Edt -> (Vork,Alt,Neu) -> Edt

ers :: Edt -> (Vork,Alt) -> Neu -> Edt

ers :: (Edt,Vork) -> (Alt,Neu) -> Edt

ers :: Edt -> Vork -> (Alt -> Neu) -> Edt

...
```

Mischformen (fgs.)

Stets gilt:

- Es wird ein Argument zur Zeit konsumiert
- Die entstehenden Funktionsterme sind (bis auf den jeweils letzten) wieder von funktionalem Wert.

Beispiel (1)

Zur Illustration betrachten wir folgendes

(Anti-) Beispiel:

```
ers :: Edt -> Vork -> (Alt -> Neu) -> Edt
```

- Beachte: Die obige Funkion `ers` erwartet an dritter Stelle ein funktionales Argument vom Typ `(Alt -> Neu)`, eine Funktion wie etwa die Funktion `copyText`:

  ```
  copyText :: Alt -> Neu
  copyText s = s ++ s
  ```

  Wir werden sehen, obige Typung ist so nicht sinnvoll!

Beispiel (2)

Zunächst erhalten wir:

- Die Funktion `ers` konsumiert ein Argument vom Typ `Edt` und der resultierende Funktionsterm ist selbst wieder eine Funktion, eine Funktion vom Typ `(Vork -> (Alt -> Neu) -> Edt)`:

```
(ers "dies ist text") :: (Vork -> (Alt -> Neu) -> Edt)
```

- Die Funktion `(ers "dies ist text")` konsumiert ein Argument vom Typ `Vork` und der resultierende Funktionsterm ist selbst wieder eine Funktion, eine Funktion vom Typ `((Alt -> Neu) -> Edt)`:

```
((ers "dies ist text") 1) :: ((Alt -> Neu) -> Edt)
```

Beispiel (3)

- Die Funktion `((ers "dies ist text") 1)` konsumiert ein Argument vom Typ `(Alt -> Neu)` und der resultierende Funktionsterm ist von einem elementaren Typ, dem Typ `Edt`:

```
(((ers "dies ist text") 1) copyText) :: Edt
```

Problem:

Der Funktionsterm

- `(((ers "dies ist text") 1) copyText)` ist bereits vom elementaren Typ `Edt`.
- Prinzipiell lieferte uns `copyText` für jede Zeichenreihe `s` die an ihrer Stelle einzusetzende Zeichenreihe `t`.
- Ein Argument `s` wird aber nicht mehr erwartet.

Diskussion des Beispiels

- Die beiden naheliegenden (?) “Rettungsversuche”

  ```
  (1) (((ers "dies ist text") 1) copyText) "abc"
  (2) (((ers "dies ist text") 1) (copyText "abc"))
  ```

  sind nicht typ-korrekt!

- In Fall (1) wenden wir
  - den Wert `(((ers "dies ist text") 1) copyText)` vom nicht-funktionalen Typ `Edt` auf eine Zeichenreihe `"abc"` vom Typ `String` (Typalias zu `Alt`, `Neu`, `Edt`) an.

- In Fall (2) wenden wir
  - den funktionalen Wert `((ers "dies ist text") 1)` vom Typ `(Alt -> Neu)` auf den elementaren Wert `(copyText "abc")` vom Typ `Neu` an.

Diskussion des Beispiels (fgs.)

Offenbar ist

```
ers :: Edt -> Vork -> (Alt -> Neu) -> Edt
```

eine nicht sinnvolle Typung im Hinblick auf unser Ziel einer Textersetzungsfunktion gewesen.

Eine

- Textersetzung findet nicht in der intendierten Weise statt.
- Die Funktion erfüllt somit nicht die mit ihr verbundene Abbildungsidee.

Zwei mögliche Abänderungen zur Abhilfe

```
ers :: Edt -> Vork -> (Alt -> Neu) -> Alt -> Edt
```

```
ers :: Edt -> [Vork] -> (Alt -> Neu) -> [Alt] -> Edt
```

Resümee

Unterschiedlich geklammerte Signaturen wie in

```
ers :: Edt -> Vork -> Alt -> Neu -> Edt

ers :: Edt -> Vork -> (Alt -> Neu) -> Edt
```

sind bedeutungsverschieden und deshalb zu unterscheiden.

Die vollständige, aber nicht überflüssige Klammerung macht die Unterschiede besonders augenfällig:

```
ers :: (Edt -> (Vork -> (Alt -> (Neu -> Edt))))

ers :: (Edt -> (Vork -> ((Alt -> Neu) -> Edt)))
```

Resümee (fgs.)

Generell gilt (in Haskell):

- Funktionssignaturen sind rechtsassoziativ geklammert.
- Funktionsterme sind linksassoziativ geklammert.
- Funktionen sind einstellig.

Daraus ergibt sich:

- Das "eine" Argument einer Haskell-Funktion ist von demjenigen Typ, der links vor dem ersten Vorkommen des Typoperators $\rightarrow$ in der Funktionssignatur steht; das "eine" Argument eines Operators in einem Funktionsterm ist der unmittelbar rechts von ihm stehende.
- Wann immer etwas anderes gemeint ist, muss dies durch explizite Klammerung in Signatur und Funktionsterm ausgedrückt werden.
- Klammern in Signaturen und Funktionstermen sind mehr als schmückendes Beiwerk; sie bestimmen die Bedeutung.

Kapitel 2.7

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 3, Funktionen und Operatoren)
- Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 1, Elemente funktionaler Programmierung)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 3, Types and Classes; Kapitel 4, Defining Functions)
- 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!)

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

- 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 3, Basic types and definitions; Kapitel 5, Data types, tuples and lists)

Kapitel 3

Rekursion

Kapitel 3.1

Rekursionstypen

Rekursion

In funktionalen Sprachen

- zentrales (Sprach-/Ausdrucks-) Mittel, Wiederholungen auszudrücken (Beachte: Wir haben keine Schleifen in funktionalen Sprachen).

Rekursion führt

- oft auf sehr elegante Lösungen, die vielfach wesentlich einfacher und intuitiver als schleifenbasierte Lösungen sind (typische Beispiele: Quicksort, Türme von Hanoi).

Insgesamt so wichtig, dass

- eine Klassifizierung von Rekursionstypen zweckmäßig ist.

$\rightsquigarrow$ eine solche Klassifizierung nehmen wir in der Folge vor.

Typische Beispiele

Sortieren mittels

- Quicksort

und die auf eine hinterindische Sage zurückgehende unter dem Namen

- Türme von Hanoi

bekannte Aufgabe einer Gruppe von Mönchen, die seit dem Anbeginn der Zeit damit beschäftigt sind, einen Turm aus 50 goldenen Scheiben mit nach oben hin abnehmendem Durchmesser umzuschichten,* sind zwei

- typische Beispiele, für die die Abstützung auf Rekursion auf intuitive, einfache und elegante Lösungen führen.

* Die Sage berichtet, dass das Ende der Welt gekommen ist, wenn die Mönche ihre Aufgabe abgeschlossen haben.

Quicksort

Erinnerung:

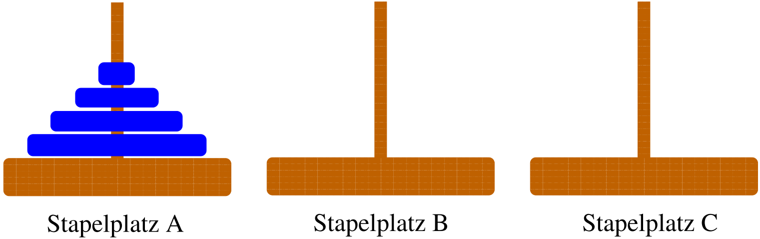
```
quickSort :: [Integer] -> [Integer]
quickSort []     = []
quickSort (x:xs) = quickSort [ y | y<-xs, y<=x ] ++
                   [x] ++
                   quickSort [ y | y<-xs, y>x ]
```

Türme von Hanoi

- Ausgangssituation:
  Gegeben sind drei Stapel(plätze) A, B und C. Auf Platz A liegt ein Stapel paarweise verschieden großer Scheiben, die von unten nach oben mit abnehmender Größe sortiert aufgeschichtet sind.
- Aufgabe:
  Verlege den Stapel von Scheiben von Platz A auf Platz C unter Zuhilfenahme von Platz B.
- Randbedingung:
  Scheiben dürfen stets nur einzeln verlegt werden und zu keiner Zeit darf eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze liegen.

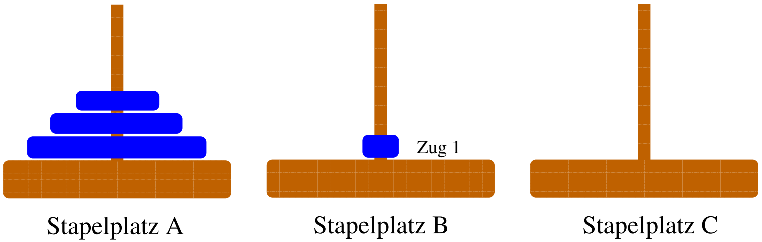
Türme von Hanoi (1)

Ausgangssituation:



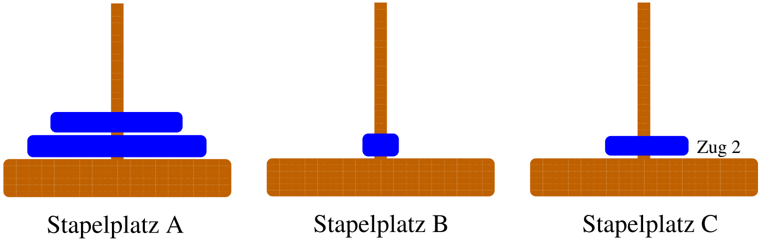
Türme von Hanoi (2)

Nach einem Zug:



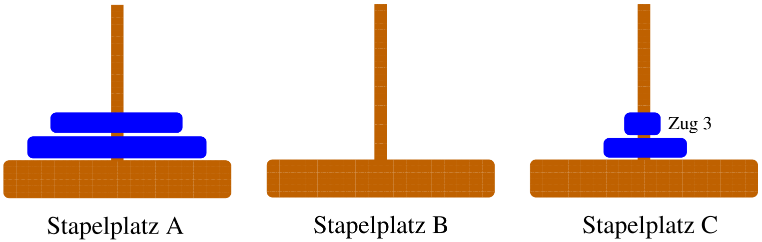
Türme von Hanoi (3)

Nach zwei Zügen:



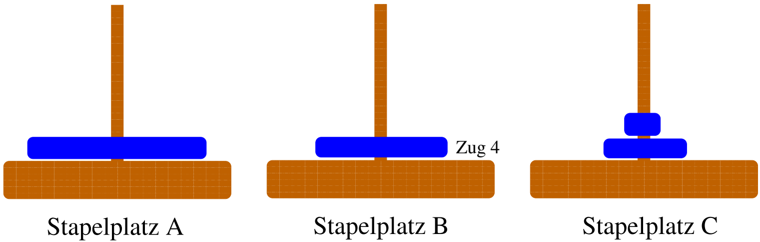
Türme von Hanoi (4)

Nach drei Zügen:



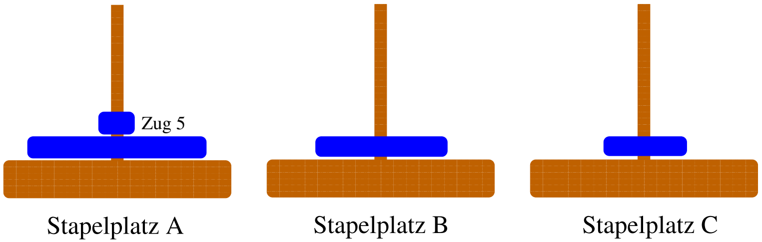
Türme von Hanoi (5)

Nach vier Zügen:



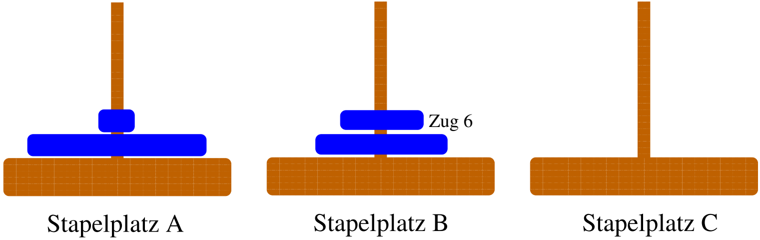
Türme von Hanoi (6)

Nach fünf Zügen:



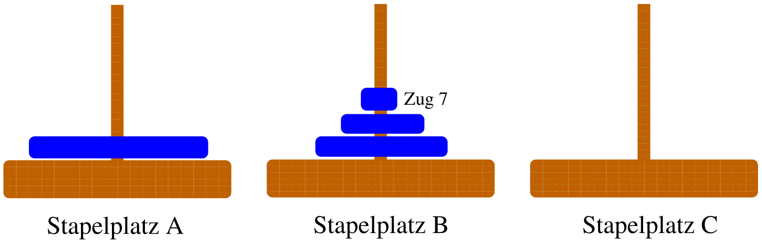
Türme von Hanoi (7)

Nach sechs Zügen:



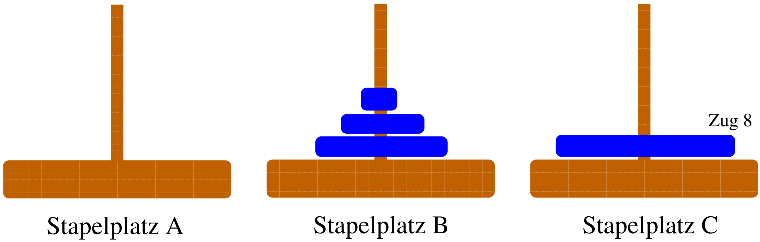
Türme von Hanoi (8)

Nach sieben Zügen:



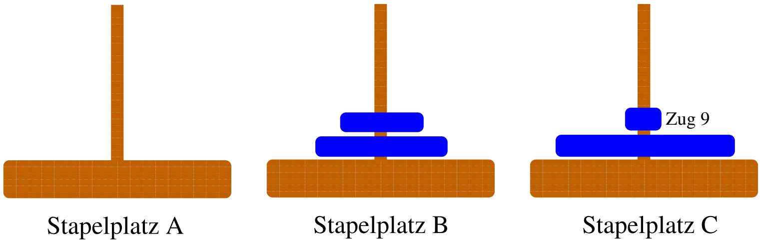
Türme von Hanoi (9)

Nach acht Zügen:



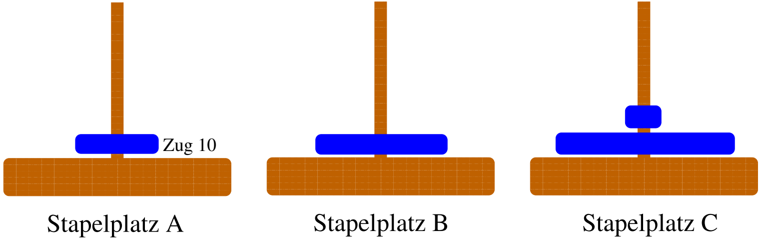
Türme von Hanoi (10)

Nach neun Zügen:



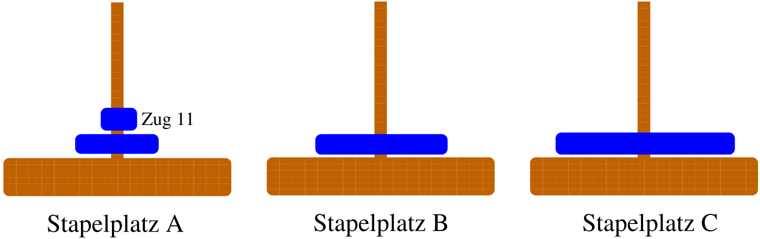
Türme von Hanoi (11)

Nach zehn Zügen:



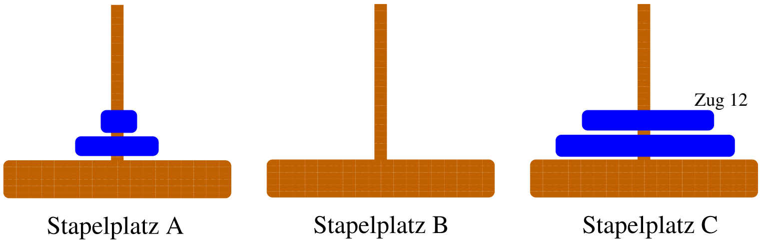
Türme von Hanoi (12)

Nach elf Zügen:



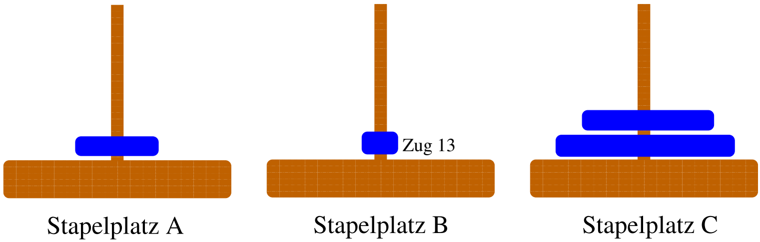
Türme von Hanoi (13)

Nach zwölf Zügen:



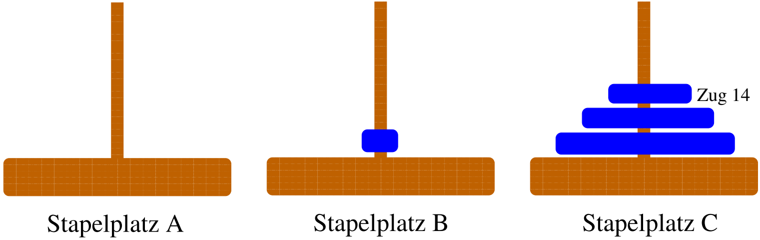
Türme von Hanoi (14)

Nach dreizehn Zügen:



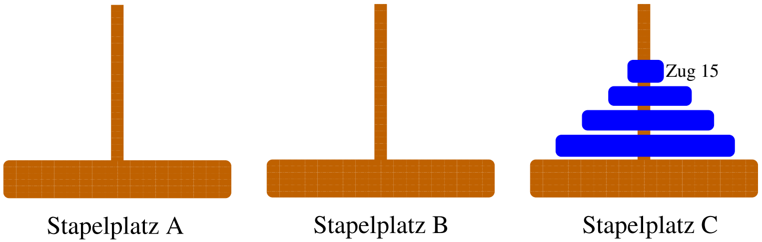
Türme von Hanoi (15)

Nach vierzehn Zügen:



Türme von Hanoi (16)

Nach fünfzehn Zügen:



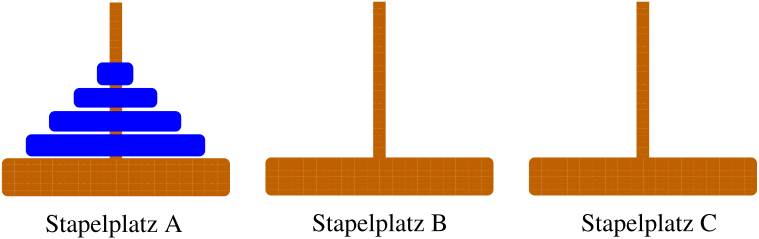
Türme von Hanoi: Rekursive Impl.Idee (1)

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 bewegen,

1) bewege den Turm $[1, 2, \ldots, N-1]$ aus $n-1$ Scheiben von A nach B unter Zuhilfenahme von Stapel C
2) bewege die nun frei liegende unterste Scheibe N von A nach C
3) bewege den Turm $[1, 2, \ldots, N-1]$ aus $n-1$ Scheiben von B nach C unter Zuhilfenahme von Stapel A

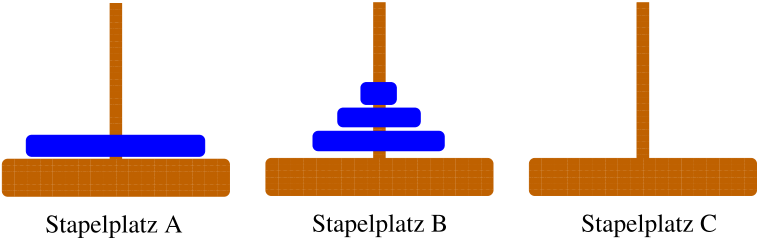
Türme von Hanoi: Rekursive Impl.Idee (2)

Aufgabe: Bewege Turm $[1, 2, \ldots, N]$ von Ausgangsstapel A auf Zielstapel C unter Verwendung von B als Zwischenlager:



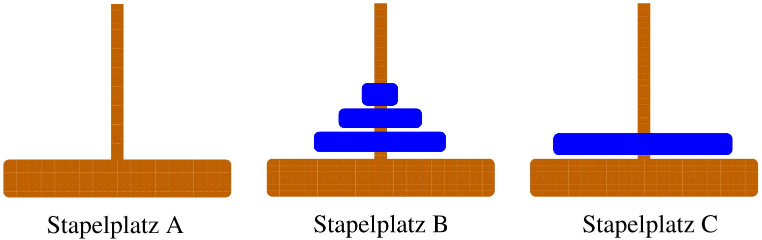
Türme von Hanoi: Rekursive Impl.Idee (3)

1) Platz schaffen & freispielen: Bewege Turm $[1, 2, \ldots, N-1]$ von Ausgangsstapel A auf Zwischenlagerstapel B:



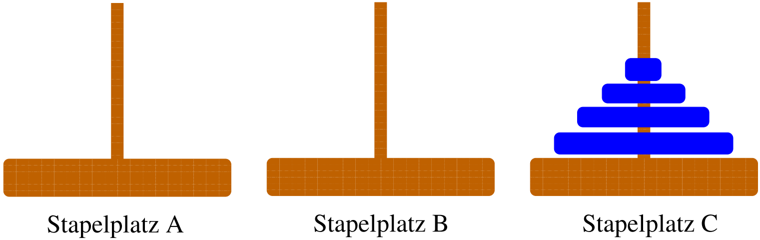
Türme von Hanoi: Rekursive Impl.Idee (4)

2) Freigespielt, jetzt wird gezogen: Bewege Turm $[N]$ (d.h. Scheibe N) von Ausgangsstapel A auf Zielstapel C:



Türme von Hanoi: Rekursive Impl.Idee (5)

3) Aufräumen: Bewege Turm $[1, 2, \ldots, N-1]$ von Zwischenlagerstapel B auf Zielstapel C:



Türme von Hanoi: Implementierung in Haskell

```
type Turmhoehe      = Int   -- Anzahl Scheiben
type ScheibenNr     = Int   -- Scheibenidentifikator
type AusgangsStapel = Char  -- Ausgangsstapel
type ZielStapel     = Char  -- Zielstapel
type HilfsStapel    = Char  -- Zwischenlagerstapel

hanoi :: Turmhoehe
          -> AusgangsStapel -> ZielStapel -> HilfsStapel
              -> [(ScheibenNr,AusgangsStapel,ZielStapel)]
hanoi n a z h
 | n==0      = []               -- Nichts zu tun, fertig
 | otherwise =
    (hanoi (n-1) a h z) ++      -- (N-1)-Turm von A nach H
    [(n,a,z)] ++                -- Scheibe N von A nach Z
    (hanoi (n-1) h z a)         -- (N-1)-Turm von H nach Z
```

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')]
```

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

Rek.typen: Mikroskopische Ebene (1)

Üblich sind folgende Unterscheidungen und Sprechweisen:

1. Repetitive (schlichte, endständige) Rekursion
   $\rightsquigarrow$ pro Zweig höchstens ein rekursiver Aufruf und zwar jeweils als äußerste Operation

   Beispiel:

   ```
   ggt :: Integer -> Integer -> Integer
   ggt m n
     | n == 0  = m
     | m >= n  = ggt (m-n) n
     | m < n   = ggt (n-m) m
   ```

Rek.typen: Mikroskopische Ebene (2)

2. Lineare Rekursion
   $\rightsquigarrow$ pro Zweig höchstens ein rekursiver Aufruf, jedoch nicht notwendig als äußerste Operation

   Beispiel:

   ```
   powerThree :: Integer -> Integer
   powerThree n
     | n == 0   = 1
     | n > 0    = 3 * powerThree (n-1)
   ```

   Beachte: Im Zweig $n > 0$ ist "$*$" die äußerste Operation, nicht powerThree!

Rek.typen: Mikroskopische Ebene (3)

3. 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`?

Rek.typen: Mikroskopische Ebene (4)

4. 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
 | otherwise    = binom (n-1,k-1) + binom (n-1,k)
```

Rek.typen: Mikroskopische Ebene (4)

Zusammenfassung:

Rekursionstypen auf der mikroskopischen Ebene

- Repetitive (schlichte, endständige) Rekursion
- Lineare Rekursion
- Geschachtelte Rekursion
- Baumartige (kaskadenartige) Rekursion

Gemeinsamer Oberbegriff

- Rekursion, präziser: Direkte Rekursion

In der Folge

- Indirekte Rekursion

Rek.typen: Makroskopische Ebene (6)

Indirekte (verschränkte, wechselweise) Rekursion
$\rightsquigarrow$ zwei oder mehr Funktionen rufen sich wechselweise auf

Beispiel:

```
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)
```

Eleganz, Effizienz, Effektivität und Implementierung

Viele Probleme lassen sich rekursiv

- elegant lösen (z.B. Quicksort, Türme von Hanoi)
- jedoch nicht immer unmittelbar effizient ($\neq$ effektiv!) (z.B. Fibonacci-Zahlen)
  - Gefahr: (Unnötige) Mehrfachberechnungen
  - Besonders anfällig: Baum-/kaskadenartige Rekursion

Vom Implementierungsstandpunkt ist

- repetitive Rekursion am (kosten-) günstigsten
- geschachtelte Rekursion am ungünstigsten

Fibonacci-Zahlen

Die Folge $f_0, f_1, f_2, \ldots$ der Fibonacci-Zahlen ist definiert durch

$$f_0 = 0, f_1 = 1 \quad \text{und} \quad f_n = f_{n-1} + f_{n-2} \quad \text{für alle } n \geq 2$$

Fibonacci-Zahlen (1)

Die naheliegende Implementierung mit baum-/kaskadenartiger Rekursion

```
fib :: Integer -> Integer
fib n
  | n == 0      = 0
  | n == 1      = 1
  | otherwise   = fib (n-1) + fib (n-2)
```

...ist sehr, seeehr langsaaaaaaaam (ausprobieren!)

Fibonacci-Zahlen (2)

Veranschaulichung ...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
```

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
```

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
```

Fibonacci-Zahlen: Schlussfolgerungen

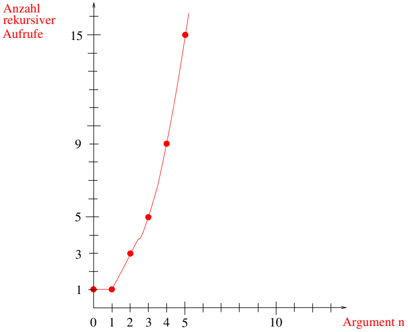
Zentrales Problem

...naiv baumartig-rekursive Berechnung der Fibonacci-Zahlen

- Sehr, sehr viele Mehrfachberechnungen

Insgesamt führt dies zu

- exponentiell wachsendem Aufwand!



Fibonacci-Zahlen effizient berechnet (1)

Fibonacci-Zahlen lassen sich auf unterschiedliche Weise effizient berechnen; z.B. mithilfe einer sog.

- Memo-Funktion

...eine Idee, die auf Donald Michie zurückgeht:

- Donald Michie. 'Memo' Functions and Machine Learning. Nature 218:19-22, 1968.

```
flist :: [Integer]
flist = [ fib n | n <- [0..] ]

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

Hinweis: Die Elementzugriffsfunktion `(!!)` hat die Signatur `(!!) :: [a] -> Int -> a`; deshalb hat `fib` hier den Argumentbereich `Int`, nicht `Integer`.

Fibonacci-Zahlen effizient berechnet (2)

Beachte: Auch ohne Memo-Listen lassen sich die Fibonacci-Zahlen effizient berechnen.

Hier ist eine Möglichkeit dafür:

- Trick: Rechnen auf Parameterposition!

```
fib :: Integer -> Integer
fib n = fib' 0 1 n where
  fib' a b 0 = a
  fib' a b n = fib' b (a+b) (n-1)
```

Zur Übung: Überlegen Sie sich, dass und wie die obige Implementierung der Funktion `fib` die Fibonacci-Zahlen berechnet.

Abhilfe bei ungünstigem Rekursionsverhalten

(Oft) ist folgende Abhilfe bei unzweckmäßigen Implementierungen möglich:

- Umformulieren!
  Ersetzen ungünstiger durch günstigere Rekursionsmuster!

  Beispiel:

  - Rückführung linearer Rekursion auf repetitive Rekursion

Rückführung linearer auf repetitive Rekursion

...am Beispiel der Fakultätsfunktion:

Naheliegende Formulierung mit

- linearer Rekursion

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

Rückführung linearer auf repetitive Rek. (fgs.)

Günstigere Formulierung mit repetitiver Rekursion:

```
facR :: (Integer,Integer) -> Integer
facR (p,r) = if p == 0 then r else facR (p-1,p*r)

fac :: Integer -> Integer
fac n = facR (n,1)
```

- Transformations-Idee: Rechnen auf Parameterposition!

Beachte: Überlagerungen mit anderen Effekten sind möglich, so dass sich möglicherweise kein Effizienzgewinn realisiert!

Andere Abhilfen

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 3.2

Komplexitätsklassen

Komplexitätsklassen (1)

Komplexitätsklassen und deren Veranschaulichung hier nach

- Peter Pepper. Funktionale Programmierung in OPAL, ML, Haskell und Gofer, Springer-V, 2. Auflage, 2003, Kapitel 11.

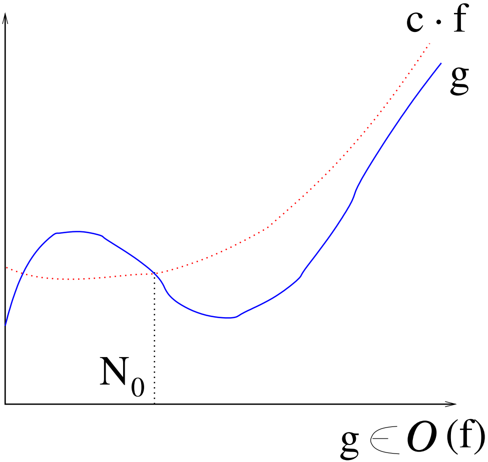
$\mathcal{O}$-Notation:

- Sei f eine Funktion $f : \alpha \to IR^+$ 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:

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Komplexitätsklassen (2)

Veranschaulichung:



Komplexitätsklassen (3)

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

Komplexitätsklassen (4)

...und eine Illustration, 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...

Fazit

Die vorigen Überlegungen machen deutlich:

- Rekursionsmuster haben einen erheblichen Einfluss auf die Effizienz einer Implementierung (siehe naive baumartig-rekursive Implementierung der Fibonacci-Funktion).
- Die Wahl eines zweckmäßigen Rekursionsmusters ist daher eminent wichtig für Effizienz!

Beachte:

- Nicht das baumartige Rekursionsmuster an sich ist ein Problem, sondern im Fall der Fibonacci-Funktion die (unnötige) Mehrfachberechnung von Werten!
- Insbesondere: Baumartig-rekursive Funktionsdefinitionen bieten sich zur Parallelisierung an!
  Stichwort: Teile und herrsche / divide and conquer / divide et impera!

Kapitel 3.3

Aufrufgraphen

Struktur von Programmen

Programme funktionaler Programmiersprachen, speziell Haskell-Programme, sind i.a.

- Systeme (wechselweiser) rekursiver Rechenvorschriften, die sich hierarchisch oder/und wechselweise aufeinander abstützen.

Um sich über die Struktur solcher Systeme von Rechenvorschriften Klarheit zu verschaffen, ist neben der Untersuchung

- der Rekursionstypen

der beteiligten Rechenvorschriften insbesondere auch die Untersuchung

- ihrer Aufrufgraphen

hilfreich.

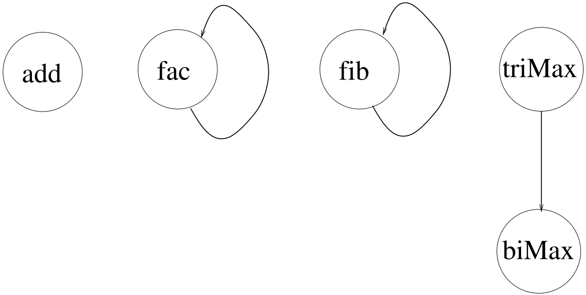
Aufrufgraphen

Der Aufrufgraph eines Systems S von Rechenvorschriften 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.

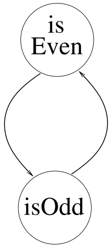
Beispiele für Aufrufgraphen (1)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `add`, `fac`, `fib`, `biMax` und `triMax`:



Beispiele für Aufrufgraphen (2)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `isOdd` und `isEven`:



Beispiele für Aufrufgraphen (3a)

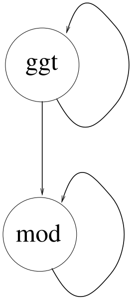
...das System von Rechenvorschriften der Funktionen ggt und mod:

```
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 für Aufrufgraphen (3b)

...und der Aufrufgraph dieses Systems:



Interpretation von Aufrufgraphen

Aus dem 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 `triMax` und `biMax`)

Interpretation von Aufrufgraphen (fgs.)

- 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 3.4

Literaturverzeichnis, Leseempfehlungen

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

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 4, Rekursion als Entwurfstechnik; Kapitel 9, Laufzeitanalyse von Algorithmen)
- 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 3 (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 3 (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)

Teil II

Applikative Programmierung

Applikatives Programmieren

...im strengen Sinn:

- Applikatives Programmieren ist ein Programmieren auf dem Niveau von elementaren Daten.
- Mit Konstanten, Variablen und Funktionsapplikationen werden Ausdrücke gebildet, die als Werte stets elementare Daten besitzen.
- 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) Argumente.

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 Funktionalen neue Funktionen gebildet.
- Es treten im Programm keine Applikationen von Funktionen auf elementare Daten auf.

Damit:

- Tragendes Konzept funktionaler Programmierung zur Programmerstellung ist die Bildung von neuen Funktionen aus gegebenen Funktionen mit Hilfe von Funktionalen.

Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 1.

Kapitel 4

Auswertung von Ausdrücken

Auswertung von Ausdrücken

...in zwei Stufen:

Auswertung von

- einfachen
- funktionalen

Ausdrücken.

Zentral für die Ausdrucksauswertung

...das Zusammenspiel von

- Expandieren ($\rightsquigarrow$ Funktionsaufrufe)
- Simplifizieren ($\rightsquigarrow$ einfache Ausdrücke)

zu organisieren, um einen Ausdruck soweit zu vereinfachen wie möglich.

Kapitel 4.1

Auswertung von einfachen Ausdrücken

Auswerten von einfachen Ausdrücken

Viele (Simplifikations-) Wege führen zum Ziel:

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 4.2

Auswertung von funktionalen Ausdrücken

Auswerten von Funktionsaufrufen (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 von Funktionsaufrufen (2)

```
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 von Funktionsaufrufen (3)

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
```

Hauptresultat (im Vorgriff auf Kap. 9 und 10)

Theorem

Jede terminierende Folge von Expansions- und Simplifikationsschritten endet mit demselben Wert.

Alonzo Church, John Barkley Rosser (1936)

Kapitel 4.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 4 (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 4 (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 5

Programmentwicklung, Programmverstehen

Kapitel 5.1

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

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 einigen 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 entsprechenden 7- bzw. 8-schritten Entwurfsprozess vor, der einen Entwurfsprozess von 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).

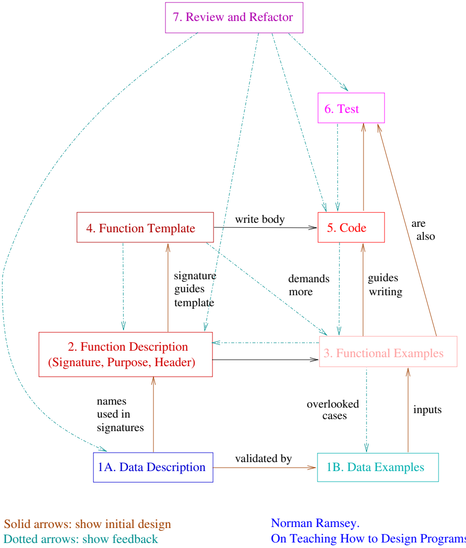
5. Vervollständige das Skelett zu einer vollständigen Funktionsimplementierung (engl. code).

6. Teste die Funktion.

7. Beurteile die Funktion und refaktorisiere sie bei Bedarf.

Verfeinerter Entwurfsprozess nach Ramsey (2)

Graphische Darstellung des Entwurfsprozesses nach Ramsey:



Norman Ramsey.
On Teaching How to Design Programs.
In Proceedings ICFP 2014, Figure 1, p. 154.

Kapitel 5.2

Programmverstehen

Motivation

Es ist eine Binsenweisheit, dass

- Programme häufiger gelesen als geschrieben werden!

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

...und die daraus ableitbaren Fragen an und Einsichten über ein Programm.

Erfolgversprechende Vorgehensweisen sind:

(1) Lesen des Programms
(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

Laufendes Beispiel

In der Folge werden wir dies im einzelnen anhand des folgenden Beispiels demonstrieren:

```
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) Programmlesen

...liefert

- allein durch Lesen der Funktionssignatur Einsichten über Art und Typ der Argumente und des Resultats; in unserem Beispiel: Die Funktion `mapWhile` erwartet als Argumente
  - eine Funktion `f` eines nicht weiter eingeschränkten Typs `a -> b`, d.h. `f :: a -> b`
  - eine Eigenschaft von Objekten vom Typ `a`, genauer ein Prädikat oder eine Wahrheitswertfunktion `p :: a -> Bool`
  - eine Liste von Elementen vom Typ `a`, d.h. eine Liste `l :: [a]`

  und liefert als Resultat
  - eine Liste von Elementen vom Typ `b`, d.h. eine Liste `l' :: [b]`
- weitere tiefergehende Einsichten durch Lesen eingestreuter Programmkommentare, auch in Form von Vor- und Nachbedingungen.

(1) Programmlesen (fgs.)

...liefert

- durch Lesen der Funktionsdefinition erste weitere Einsichten über das Verhalten und die Bedeutung des Programms; in unserem 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

...liefert

- tiefere Einsichten über Programmverhalten und -bedeutung, auch durch den Beweis von Eigenschaften, die das Programm besitzt; in unserem Beispiel etwa können wir für alle Funktionen `f`, Prädikate `p` und endliche Listen `xs` 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 etwa (mW5) und (mW6) Folgerungen aus (mW4) sind.

(3) Gedankliche oder Papier- und Bleistift-

...Ausführung des Programms hilft

- Verhaltenshypothesen zu validieren oder zu generieren durch Berechnung der Funktionswerte für ausgewählte Argumente, z.B.:

```
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 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)

Der Lesbarkeit und Verstehbarkeit eines Programms sollte

- immer schon beim Schreiben des Programms Bedacht gezollt werden, nicht zuletzt im höchst eigenen Interesse!

Kapitel 5.3

Literaturverzeichnis, Leseempfehlungen

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

Kapitel 6

Datentypdeklarationen

Grundlegende Datentypstrukturen

...in Programmiersprachen sind:

- Aufzählungstypen
- Produkttypen
- Summentypen

Typische Beispiele d. grundlegenden Typmuster

- Aufzählungstypen
  ⇝ Typen mit endlich vielen Werten

  Typisches Beispiel: Typ `Jahreszeiten` mit Werten `Fruehling`, `Sommer`, `Herbst` und `Winter`.

- Produkttypen (synonym: Verbundtypen, "record"-Typen)
  ⇝ Typen mit möglicherweise unendlich vielen Tupelwerten

  Typisches Beispiel: Typ `Person` mit Werten `(Adam, maennlich, 23)`, `(Eva, weiblich, 21)`, etc.

- Summentypen (synonym: Vereinigungstypen)
  ⇝ Vereinigung von Typen mit möglicherweise jeweils unendlich vielen Werten

  Typisches Beispiel: Typ `Medien` als Vereinigung der (Werte der) Typen `Buch`, `E-Buch`, `DVD`, `CD`, etc.

Datentypdeklarationen in Haskell

Haskell bietet

1. Algebraische Datentypen (`data Tree = ...`)

als einheitliches Konzept zur Spezifikation von

- Aufzählungstypen, Produkt- und Summentypen

an.

Zusätzlich bietet Haskell zwei davon zu unterscheidende verwandte Sprachkonstrukte an:

2. Typsynonyme (`type Student = ...`)
3. Typidentitäten (`newtype State = ...`) als Datentypdeklaration eingeschränkter Art

In der Folge

...werden wir diese Sprachkonstrukte, ihre Gemeinsamkeiten, Unterschiede und Anwendungskontexte im Detail untersuchen.

Kapitel 6.1

Typsynonyme

Motivation

Die Deklaration einer Selektorfunktion mit Signatur

```
titel :: (String,String,(Int,Int,Int)) -> String
titel (t, i, (h,m,s)) = t
```

ist trotz des “sprechenden” Namens vergleichsweise nichtssagend.

Typsynonyme können hier auf einfache Weise Abhilfe schaffen!

Typsynonyme im Beispiel (1)

Deklariere:

```
type Titel      = String
type Interpret  = String
type Spieldauer = (Int,Int,Int)
```

Diese Typsynonyme erlauben nun folgende Signatur:

```
titel :: (Titel,Interpret,Spieldauer) -> Titel
titel (t, i, (h,m,s)) = t

interpret ::
  (Titel,Interpret,Spieldauer) -> Interpret
interpret (t, i, (h,m,s)) = i

spieldauer ::
  (Titel,Interpret,Spieldauer) -> Spieldauer
spieldauer (t, i, (h,m,s)) = (h,m,s)
```

Typsynonyme im Beispiel (2)

Deklariere:

```
type Titel       = String
type Interpret   = String
type Spieldauer = (Int,Int,Int)
type CD          = (Titel,Interpret,Spieldauer)
```

Mit diesen Typsynonymen werden folgende Signaturen der Selektorfunktionen mögich:

```
titel :: CD -> Titel
titel (t, i, (h,m,s)) = t

interpret :: CD -> Interpret
interpret (t, i, (h,m,s)) = i

spieldauer :: CD -> Spieldauer
spieldauer (t, i, (h,m,s)) = (h,m,s)
```

Typsynonyme im Beispiel (3)

Deklariere:

```
type Titel       = String
type Regisseur   = String
type Spieldauer = (Int,Int,Int)
type DVD         = (Titel,Regisseur,Spieldauer)
```

Hier erlauben die Typsynonyme folgende Signaturen der Selektorfunktionen:

```
titel :: DVD -> Titel
titel (t, r, (h,m,s)) = t

regisseur :: DVD -> Regisseur
regisseur (t, r, (h,m,s)) = r

spieldauer :: DVD-> Spieldauer
spieldauer (t, r, (h,m,s)) = (h,m,s)
```

Typsynonyme im Überblick

Die Deklaration von Typsynonymen wird durch

- das Schlüsselwort `type` eingeleitet.

Dabei ist unbedingt zu beachten:

- `type` führt neue Namen für bereits existierende Typen ein (Typsynonyme!), keine neuen Typen.

Typsynonyme

- führen daher nicht zu (zusätzlicher) Typsicherheit!

Keine zusätzliche Typsicherheit durch type (1)

Betrachte:

```
cd  = ("Alpengluehn","Hansi Hinterseer",(1,8,36)) :: CD
dvd = ("Der Bockerer","Franz Antel",(2,7,24)) :: DVD
```

Erwartungsgemäß erhalten wir:

```
interpret cd  ->> "Hansi Hinterseer"
regisseur dvd ->> "Franz Antel"
```

Wir erhalten aber auch:

```
interpret dvd ->> "Franz Antel"
regisseur cd  ->> "Hansi Hinterseer"
```

trotz der anscheinend widersprechenden Signaturen:

```
interpret :: CD -> Interpret
regisseur :: DVD -> Regisseur
```

Keine zusätzliche Typsicherheit durch type (2)

Der Grund:

- `CD`, `DVD`, `Interpret` und `Regisseur` bezeichnen Typsynonyme, keine eigenständigen Typen.

```
type Titel      = String
type Interpret  = String
type Regisseur  = String
type Spieldauer = (Int,Int,Int)
type CD         = (Titel,Interpret,Spieldauer)
type DVD        = (Titel,Regisseur,Spieldauer)
```

Die durch die Synonyme `CD` und `DVD` bezeichneten Typen unterscheiden sich durch nichts von dem durch das Synonym `Tripel` bezeichneten Typ:

```
type Tripel     = (String,String,(Int,Int,Int))
```

Keine zusätzliche Typsicherheit durch type (3)

Wo immer ein Wert vom Typ

- `(String,String,(Int,Int,Int))`

stehen darf, darf auch ein Wert der Typsynonyme

- `CD`, `DVD` und `Tripel`

stehen (und umgekehrt)!

Genauso wie auch ein Wert vom Typ

- `([Char],[Char],(Int,Int,Int))`

wg.

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

Ein (gar nicht so) pathologisches Beispiel

```
type Euro        = Float
type Yen         = Float
type Temperature = Float

myPi   :: Float
myPi   = 3.14
daumen :: Float
daumen = 5.55
tmp    :: Temperature
tmp    = 43.2

currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

Mit obigen Deklarationen:

```
currencyConverter maxTemp ->> 60.627
```

werden 43.2 °C in 60.627 Yen umgerechnet. Typsicher? Nein!

Ein im Kern reales Beispiel (1)

Anflugsteuerung einer Sonde zum Mars:

```
type Meilen          = Float
type Km              = Float
type Zeit            = Float
type Geschwindigkeit = Float
type Wegstrecke      = Meilen
type Abstand         = Km

geschwindigkeit :: Wegstrecke -> Zeit -> Geschwindigkeit
geschwindigkeit w z = (/) w z

verbleibendeFlugzeit :: Abstand -> Geschwindigkeit -> Zeit
verbleibendeFlugzeit a g = (/) a g
```

Ein im Kern reales Beispiel (2)

```
abs  = 18524.34 :: Abstand
weg  = 1117.732 :: Meilen
zeit = 937.2712 :: Zeit

verbleibendeFlugzeit abs (geschwindigkeit weg zeit)
```

???

...durch Typisierungsprobleme dieser Art (Abstand in km; Geschwindigkeit in Meilen pro Sekunde) ging vor einigen Jahren eine Sonde im Wert von mehreren 100 Mill. USD am Mars verloren.

Resümee

Durch `type`-Deklarationen eingeführte Typsyonyme

- tragen zur Dokumentation bei
- erleichtern (bei treffender Namenswahl) das Programmverständnis
- sind sinnvoll, wenn mehrere Typen durch denselben Grundtyp implementiert werden

Aber:

- Typsynonyme führen nicht zu (zusätzlicher) Typsicherheit!

Kapitel 6.2

Neue Typen (eingeschränkter Art)

Motivation (1)

Wie lässt sich das Marssondendebakel verhindern?

Durch Verwendung eigenständiger neuer Typen:

```
newtype Meilen  = Mei Float
newtype Km      = Km  Float

type Wegstrecke = Meilen
type Abstand    = Km

newtype Zeit             = Sek Float
newtype Geschwindigkeit = MeiProSek Float
```

Motivation (2)

```
geschwindigkeit :: Wegstrecke -> Zeit -> Geschwindigkeit
geschwindigkeit (Mei w) (Sek s) = MeiProSek ((/) w s)

verbleibendeFlugzeit :: Abstand -> Geschwindigkeit -> Zeit
verbleibendeFlugzeit (Km a) (MeiProSek g) =
           -- Sek ((/) a g) ist offensichtlich falsch!
           -- 1km entspricht 1.6093472 amerik. Meilen.
           -- Deshalb folgende Implementierung:
           Sek ((/) a (g*1.6093472))

abs  = Km  18524.34 :: Abstand
weg  = Mei 1117.732 :: Meilen
zeit = Sek 937.2712 :: Zeit
```

Der Aufruf

```
verbleibendeFlugzeit abs (geschwindigkeit weg zeit)
```

...liefert jetzt die Restflugzeit korrekt!

Neue Typen im Überblick

...eingeführt mithilfe der `newtype`-Deklaration:

Beispiele:

```
newtype Meilen = Mei Float
newtype Km     = Km  Float

newtype Person = Pers (Name,Geschlecht,Alter)
newtype TelNr  = TNr Int

newtype PersVerz = PV [Person]
newtype TelVerz  = TV [(Person,TelNr)]
```

`Mei`, `Km`, `Pers`, `TNr`, `PV` und `TV` sind sog.

- (einstellige) Konstruktoren

Resümee

Durch `newtype`-Deklarationen eingeführte Typen

- sind eigenständige neue Typen
- sind typsicher und erhöhen damit die Typsicherheit im Programm
- sind sinnvoll, wenn der zugrundeliegende Typ ausdrücklich vom neu definierten Typ unterschieden werden soll

Aber:

- Durch `newtype` eingeführte Typen dürfen (anders als algebraische Datentypen) nur einen Konstruktor haben, der einstellig sein muss
- In diesem Sinn erlaubt `newtype` nur neue Typen einer eingeschränkten Art einzuführen

Kapitel 6.3

Algebraische Datentypen

Algebraische Datentypen

...sind Haskells Vehikel zur uneingeschränkten Spezifikation selbstdefinierter neuer Datentypen.

Algebraische Datentypen erlauben uns zu definieren:

- Summentypen

Als spezielle Summentypen können definiert werden:

- Produkttypen
- Aufzählungstypen

Haskell bietet damit ein

- einheitliches Sprachkonstrukt zur Definition von Summen-, Produkt- und Aufzählungstypen.

Beachte: Viele andere Programmiersprachen, z.B. Pascal, sehen dafür jeweils eigene Sprachkonstrukte vor (vgl. Anhang C).

Algebraische Datentypen in Haskell

In der Folge geben wir einige Beispiele für

- Aufzählungstypen
- Produkttypen
- Summentypen

als

- algebraische Datentypen

in Haskell an.

Entsprechende Pascal-Datentypdeklarationen sind zum Vergleich in Anhang C zusammengefasst.

Aufzählungstypen als algebraische Datentypen:

Fünf Beispiele für Aufzählungstypen

```
data Jahreszeiten = Fruehling | Sommer
                  | Herbst | Winter
data Wochenende   = Samstag | Sonntag
data Geofigur     = Kreis | Rechteck
                  | Quadrat | Dreieck
data Medien       = Buch | E-Buch | DVD | CD
```

Ebenso ein (algebraischer) Aufzählungstyp in Haskell ist der Typ der Wahrheitswerte:

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

Produkttypen als algebraische Datentypen

Zwei Beispiele für Produkttypen:

```
data Person = Pers Vorname Nachname Geschlecht Alter
data Anschrift = Adr Strasse Stadt PLZ Land

type Vorname    = String
type Nachname   = String
data Geschlecht = Maennlich | Weiblich
type Alter      = Int
type Strasse    = String
type Stadt      = String
type PLZ        = Int
type Land       = String
```

Summentypen als algebraische Datentypen (1)

Zwei Beispiele für Summentypen:

```
data Multimedien
       = Buch Autor Titel Lieferbar
         | E-Buch Autor Titel LizenzBis
         | DVD Titel Regisseur Spieldauer Untertitel
         | CD Interpret Titel Komponist

type Autor        = String
type Titel        = String
type Lieferbar    = Bool
type LizenzBis    = Int
type Regisseur    = String
type Spieldauer   = Float
type Untertitel   = Bool
type Interpret    = String
type Komponist    = String
```

Summentypen als algebraische Datentypen (2)

```
data GeometrischeFigur
        = Kreis Radius
          | Rechteck Breite Hoehe
          | Quadrat Seitenlaenge Diagonale
          | Dreieck Seite1 Seite2 Seite3 Rechtwinklig


type Radius       = Float
type Breite       = Float
type Hoehe        = Float
type Seitenlaenge = Double
type Diagonale    = Double
type Seite1       = Double
type Seite2       = Double
type Seite3       = Double
type Rechtwinklig = Bool
```

Typsynonyme bringen Transparenz!

Vergleiche:

Ohne Typsynonyme

```
data Anschrift = Adr String String Int String

data Multimedien
         = Buch String String Bool
           | E-Buch String String Int
           | DVD String String Float Bool
           | CD String String String

data GeometrischeFigur
         = Kreis Float
           | Rechteck Float Float
           | Quadrat Double Double
           | Dreieck Double Double Double Bool
```

...sind die Deklarationen nichtssagend!

Algebraische Datentypen in Haskell

...das allg. Muster der algebraischen Datentypdefinition:

```
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 ...Typname/-identifikator
- $\texttt{Con}_i$, $i = 1..n$ ...Konstruktor(en)/-identifikatoren
- k_i, $i = 1..n$ ...Stelligkeit des Konstruktors $\texttt{Con}_i$, $\mathrm{k}_i \geq 0$, $i = 1, \ldots, n$

Beachte: Typ- und Konstruktoridentifikatoren müssen mit einem Großbuchstaben beginnen (siehe z.B. True, False)!

Konstruktoren

...können als Funktionsdefinitionen gelesen werden:

$\texttt{Con}_i$ :: $\texttt{t}_{i1}$ -> ... -> $\texttt{t}_{ik_i}$ -> Typename

Die Konstruktion von Werten eines algebraischen Datentyps erfolgt durch Anwendung eines Konstruktors auf Werte "passenden" Typs, d.h.

$$\texttt{Con}_i\ \texttt{v}_{i1}\ \ldots\ \texttt{v}_{ik_i} :: \texttt{Typname}$$
$$\text{mit } \texttt{v}_{i1} :: \texttt{t}_{i1}\ \ldots\ \texttt{v}_{ik_i} :: \texttt{t}_{ik_i},\ j = 1, \ldots, k_i$$

Beispiele:

- `Pers "Adam" "Riese" Maennlich 67 :: Person`
- `Buch "Nestroy" "Der Talisman" True :: Multimedien`
- `CD "Mutter" "Variationen" "Bach" :: Multimedien`
- ...

Zwischenfazit

Aufzählungstypen, Produkttypen, Summentypen

- In Haskell: ein einheitliches Sprachkonstrukt
  $\rightsquigarrow$ die algebraische Datentypdefinition
- In anderen Sprachen, z.B. Pascal: drei verschiedene Sprachkonstrukte (vgl. Anhang C)

In der Folge fassen wir dies noch einmal systematisch zusammen.

Aufzählungstypen in Haskell

Mehrere ausschließlich nullstellige Konstruktoren führen auf Aufzählungstypen:

Beispiele:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
data Wochenende = Sonnabend | Sonntag
data Geschlecht = Maennlich | Weiblich
data Geofigur   = Kreis | Rechteck
                    | Quadrat | Dreieck
data Medien     = Buch | E-Buch | DVD | CD
```

Wie bereits festgestellt, ist insbesondere auch der Typ `Bool` der Wahrheitswerte

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

Beispiel eines in Haskell bereits vordefinierten Aufzählungstyps.

Funktionsdefinitionen über Aufzählungstypen

...üblicherweise mit Hilfe von Musterpassung (engl. pattern matching).

Beispiele:

```
hatEcken :: Form -> Bool
hatEcken Kreis = False
hatEcken _     = True

hatAudioInformation :: Multimedien -> Bool
hatAudioInformation DVD = True
hatAudioInformation CD  = True
hatAudioInformation _   = False
```

Produkttypen in Haskell

Alternativenlose mehrstellige Konstruktoren führen auf Produkttypen:

Beispiel:

```
data Person = Pers Vorname Nachname Geschlecht Alter

type Vorname    = String
type Nachname   = String
type Alter      = Int
data Geschlecht = Maennlich | Weiblich
```

Beispiele für Werte des Typs `Person`:

```
Pers "Paul" "Pfiffig" Maennlich 23  :: Person
Pers "Paula" "Plietsch" Weiblich 22 :: Person
```

Beachte: Die Funktionalität der Konstruktorfunktion `Pers` ist

```
Pers :: Vorname -> Nachname ->
                     Geschlecht -> Alter -> Person
```

Summentypen in Haskell (1)

Mehrere (null- oder mehrstellige) Konstruktoren führen auf Summentypen:

Beispiel:

```
data XGeoFigur
 = XKreis XRadius
   | XRechteck XBreite XHoehe
   | XQuadrat XSeitenlaenge XDiagonale
   | XDreieck XSeite1 XSeite2 XSeite3 XRechtwinklig
   | XEbene
```

Beachte: Die Varianten einer Summe werden durch "|" voneinander getrennt.

Summentypen in Haskell (2)

mit

```
type XRadius        = Float
type XBreite        = Float
type XHoehe         = Float
type XSeitenlaenge  = Double
type XDiagonale     = Double
type XSeite1        = Double
type XSeite2        = Double
type XSeite3        = Double
type XRechtwinklig  = Bool
```

Summentypen in Haskell (3)

Beispiele für Werte des Typs erweiterte Figur `XGeoFigur`:

```
Kreis 3.14                    :: XGeoFigur
Rechteck 17.0 4.0             :: XGeoFigur
Quadrat 47.11 66.62           :: XGeoFigur
Dreieck 3.0 4.0 5.0 True      :: XGeoFigur
Ebene                         :: XGeoFigur
```

Transparente und sprechende Datentyp-deklarationen

Bisher haben wir Typsynomyme verwendet, um für Datentypen mit mehreren Feldern transparente und sprechende Datentyp-deklarationen zu erhalten.

Grundsätzlich bietet Haskell drei Möglichkeiten dafür an:

- Transparenz durch Typsynonyme
- Transparenz durch Kommentierung
- Transparenz durch Verbundtyp-Syntax (Record-Syntax)

Sprechende Datentypdeklarationen (1)

Variante 1: Transparenz durch Typsynonyme

Bereits besprochen.

Variante 2: Transparenz durch Kommentierung

```
data PersDaten = PD
       String     -- Vorname
       String     -- Nachname
       Geschlecht -- Geschlecht (m/w)
       Int        -- Alter
       String     -- Strasse
       String     -- Stadt
       Int        -- PLZ
       String     -- Land

data Geschlecht = Maennlich | Weiblich
```

Sprechende Datentypdeklarationen (2)

(Musterdefinierte) Zugriffsfunktionen:

```
getGivenName :: PersDaten -> String
getGivenName (PD vn _ _ _ _ _ _ _) = vn

setGivenName :: String -> PersDaten -> PersDaten
setGivenName vn (PD _ nn gs al str st pz ld)
                    = PD vn nn gs al str st pz ld

createPDwithGivenName :: String -> PersDaten
createPDwithGivenName vn
       = (PD vn undefined undefined undefined
                undefined undefined undefined
                undefined)
```

Analog sind Zugriffsfunktionen für alle anderen Felder von PersDaten zu definieren. Umständlich!
Vorteilhaft: Verbundtyp-Syntax!

Sprechende Datentypdeklarationen (3)

Variante 3: Transparenz durch Verbundtyp-Syntax

```
data PersDaten = PD {
       vorname    :: String,
       nachname   :: String,
       geschlecht :: Geschlecht,
       alter      :: Int,
       strasse    :: String,
       stadt      :: String,
       plz        :: Int,
       land       :: String }

data Geschlecht = Maennlich | Weiblich
```

Sprechende Datentypdeklarationen (4)

Transparenz durch Verbundtyp-Syntax unter Zusammenfassung typgleicher Felder:

```
data PersDaten = PD {
       vorname,
       nachname,
       strasse,
       stadt,
       land       :: String,
       geschlecht :: Geschlecht,
       alter,
       plz        :: Int }

data Geschlecht = Maennlich | Weiblich
```

Sprechende Datentypdeklarationen (5)

(Musterdefinierte) Zugriffsfunktionen:

Drei gleichwertige Varianten einer Funktion, die den vollen Namen einer Person liefert:

```
fullName1 :: PersDaten -> String
fullName1 (PD vn nn _ _ _ _ _ _) = vn ++ nn

fullName2 :: PersDaten -> String
fullName2 pd = vorname pd ++ nachname pd

fullName3 :: PersDaten -> String
fullName3 (PD {vorname=vn, nachname=nn}) = vn + nn
```

Sprechende Datentypdeklarationen (6)

(Musterdefinierte) Zugriffsfunktionen (fgs.):

```
setFullName
      :: String -> String -> PersDaten -> PersDaten
setFullName vn nn pd
      = pd {vorname=vn, nachname=nn}

createPDwithFullName
      :: String -> String -> PersDaten
createPDwithFullName vn nn
      = PD vorname=vn, nachname=nn
        -- alle übrigen Felder werden automatisch
        -- "undefined" gesetzt.
```

Sprechende Datentypdeklarationen (7)

Feldnamen dürfen wiederholt werden, wenn ihr Typ gleich bleibt:

```
data PersDaten = PD {
       vorname,
       nachname,
       strasse,
       stadt,
       land       :: String,
       geschlecht :: Geschlecht,
       alter,
       plz        :: Int }
       | KurzPD {
       vorname,
       nachname   :: String }

data Geschlecht = Maennlich | Weiblich
```

Sprechende Datentypdeklarationen (8)

Vorteile der Verbundtyp-Syntax:

- Transparenz durch Feldnamen
- Automatische Generierung von Zugriffsfunktionen für jedes Feld

```
vorname :: PersDaten -> String
vorname (PD vn _ _ _ _ _ _ _)  = vn
vorname (KurzPD vn _)          = vn
...
strasse :: PersDaten -> String
strasse (PD _ _ str _ _ _ _ _) = str
...
plz :: PersDaten -> Int
plz (PD _ _ _ _ _ _ _ pz)      = pz
```

⇝ Einsparung von viel Schreibarbeit!

Resümee

Zusammenfassend ergibt sich somit die eingangs genannte Taxonomie algebraischer Datentypen:

Haskell offeriert

- Summentypen

mit den beiden Spezialfällen

- Produkttypen
  $\rightsquigarrow$ nur ein Konstruktor, i.a. mehrstellig
- Aufzählungstypen
  $\rightsquigarrow$ ein oder mehrere Konstruktoren, alle nullstellig

In der Folge betrachten wir Erweiterungen obiger Grundfälle.

Rekursive Typen (1)

...sind der Schlüssel zu (potentiell) unendlichen Datenstrukturen in Haskell.

Technisch:

...zu definierende Typnamen können rechtsseitig in der Definition benutzt werden.

Beispiel: (Arithmetische) Ausdrücke

```
data Expr = Opd Int
          | Add Expr Expr
          | Sub Expr Expr
          | Squ Expr
```

Rekursive Typen (2)

Beispiele für Ausdrücke (lies <=> als "entspricht").

```
Opd 42 :: Expr                         <=>  42
Add (Opd 17) (Opd 4) :: Expr           <=>  17+4
Add (Squ (Sub (Opd 42) (Squ (2)))) (Opd 12) :: Expr
                         <=> square(42-square(2))+12
```

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

Rekursive Typen (3)

Weitere Beispiele rekursiver Datentypen:

Binärbäume, hier zwei verschiedene Varianten:

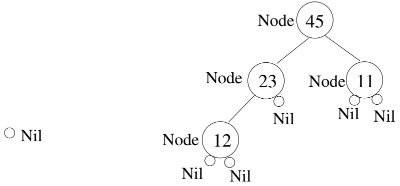
```
data BinTree1 = Nil
              | Node Int BinTree1 BinTree1

data BinTree2 = Leaf Int
              | Node Int BinTree2 BinTree2
```

Rekursive Typen (4)

Veranschaulichung der Binärbaumvarianten 1&2 anhand eines Beispiels:

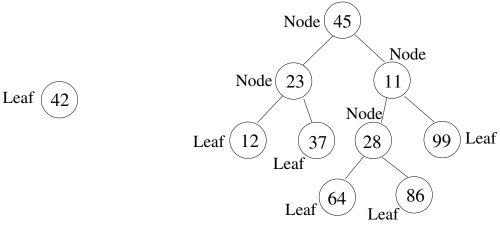
Variante 1



"Leerer" Baum

Nichtleerer Baum

Variante 2



Rekursive Typen (5)

Beispiele für (musterdefinierte) Fkt. über Binärbaumvariante 1:

```
valBT1 :: BinTree1 -> Int
valBT1 Nil                = 0
valBT1 (Node n bt1 bt2) = n + valBT1 bt1
                            + valBT1 bt2

depthBT1  :: BinTree1 -> Int
depthBT1 Nil = 0
depthBT1 (Node _ bt1 bt2)
    = 1 + max (depthBT1 bt1) (depthBT1 bt2)
```

Mit diesen Definitionen sind Beispiele gültiger Aufrufe:

```
valBT1 Nil                                ->>  0
valBT1 (Node 17 Nil (Node 4 Nil Nil))   ->> 21
depthBT1 (Node 17 Nil (Node 4 Nil Nil)) ->>  2
depthBT1 Nil                              ->>  0
```

Rekursive Typen (6)

Beispiele für (musterdefinierte) Fkt. über Binärbaumvariante 2:

```
valBT2 :: BinTree2 -> Int
valBT2 (Leaf n)          = n
valBT2 (Node n bt1 bt2) = n + valBT2 bt1
                            + valBT2 bt2

depthBT2  :: BinTree2 -> Int
depthBT2 (Leaf _) = 1
depthBT2 (Node _ bt1 bt2)
    = 1 + max (depthBT2 bt1) (depthBT2 bt2)
```

Mit diesen Definitionen sind Beispiele gültiger Aufrufe:

```
valBT2 (Leaf 3)   ->>  3
valBT2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5)))
                        ->> 42
depthBT2 (Node 17 (Leaf 4) (Node 4 (Leaf 12) (Leaf 5)))
                        ->>  3
depthBT2 (Leaf 3) ->>  1
```

Wechselweise rekursive Typen

...ein Spezialfall rekursiver Typen.

Beispiel:

```
data Individual = Adult Name Address Biography
                | Child Name

data Biography  = Parent CV [Individual]
                | NonParent CV

type CV         = String
```

Ausblick auf weitere Erweiterungen

Polymorphe Typen, sowie polymorphe und überladene Operatoren und Funktionen in Kapitel 8!

Kapitel 6.4

Zusammenfassung und Anwendungsempfehlung

Rückblick auf Datentypdeklarationen in Haskell

Haskell bietet zur Deklaration von Datentypen 3 Sprachkonstrukte an:

- `type Student = ...`: Typsynonyme
- `newtype State = ...`: Typidentitäten als eingeschränkte Variante algebraischer Datentypen
- `data Tree = ...`: Algebraische Datentypen

Es gilt:

- `type` erlaubt einen neuen Namen (einen Alias-Namen) für einen bereits existierenden Typ einzuführen, keine neuen Typen $\leadsto$ unterstützt Transparenz
- `newtype` erlaubt einem bereits existierenden Typ eine eigene neue Identität zu geben $\leadsto$ liefert Typsicherheit
- `data` (und nur `data`) erlaubt uneingeschränkt neue Datentypen einzuführen $\leadsto$ ermöglicht neue Typen

Kapitel 6.4.1

Produkttypen vs. Tupeltypen

Produkttypen vs. Tupeltypen (1)

Der Typ `Person` als

- Produkttyp

```
data Person
       = Pers Vorname Nachname Geschlecht Alter
```

- Tupeltyp

```
type Person
       = (Vorname, Nachname, Geschlecht, Alter)
```

Vordergründiger Unterschied:
...in der Tupeltypvariante fehlt gegenüber der Produkttyp-variante der Konstruktor (in diesem Bsp.: `Pers`)

Produkttypen vs. Tupeltypen (2)

Eine Abwägung von Vor- und Nachteilen:

Produkttypen und ihre typischen

- Vorteile gegenüber Tupeltypen
  - Objekte des Typs sind mit dem Konstruktor "markiert" (trägt zur Dokumentation bei)
  - Tupel mit zufällig passenden Komponenten sind nicht irrtümlich als Elemente des Produkttyps manipulierbar (Typsicherheit! Vgl. frühere Beispiele zur Umrechnung von Euro in Yen und zum Verlust der Marssonde!)
  - Aussagekräftigere (Typ-) Fehlermeldungen sind möglich (Typsynonyme können wg. Expansion in Fehlermeldungen fehlen).

Produkttypen vs. Tupeltypen (3)

- Nachteile gegenüber Tupeltypen
  - Produkttypelemente sind weniger kompakt, erfordern längere Definitionen (mehr Schreibarbeit)
  - Auf Tupeln vordefinierte polymorphe Funktionen (z.B. `fst, snd, zip, unzip,...`) stehen nicht zur Verfügung.
  - Der Code ist durch "ein-" und "auspacken" (geringfügig) weniger effizient.

Zentral: Produkttypen bieten Typsicherheit!

Mit Produkttypen statt Typsynoymen

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = TMP Float
```

```
myPi    :: Float
myPi     = 3.14
daumen  :: Float
daumen   = 5.55
maxTmp :: Temperature
maxTmp  = Tmp 43.2
```

ist ein Aufruf wie

```
currencyConverter maxTmp ->> currencyConverter (TMP 43.2)
```

wg. der Typsicherheit durch das Typsystem von Haskell (hier: fehlschlagende Musterpassung) verhindert:

```
currencyConverter :: Euro -> Yen
currencyConverter (EUR x) = YEN (x + myPi * daumen)
```

Resümee (1)

...dieser Überlegungen:

- Typsynonyme wie

```
type Euro        = Float
type Yen         = Float
type Temperature = Float
```

  ...erben alle Operationen von `Float` und sind damit beliebig austauschbar – mit allen Annehmlichkeiten und Gefahren, sprich Fehlerquellen.

- Produkttypen wie

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = TMP Float
```

  ...erben keinerlei Operationen von `Float`, bieten aber dafür um den Preis geringer zusätzlicher Schreibarbeit (und Performanzverlusts) Typsicherheit!

Resümee (2)

In ähnlicher Weise:

```
data Meilen     = Mei Float
data Km         = Km Float
type Abstand    = Meilen
type Wegstrecke = Km
...
```

...wäre auch der Verlust der Marssonde vermutlich vermeidbar gewesen.

Beachte:

- Typ- und Konstruktornamen dürfen in Haskell übereinstimmen (siehe z.B. `data Km = Km Float`)
- Konstruktornamen müssen global (d.h. modulweise) eindeutig sein.

Kapitel 6.4.2

Typsynonyme vs. Neue Typen

Es gilt (1)

`newtype`-Deklarationen verhalten sich im Hinblick auf

- Typsicherheit
  ...wie `data`-Deklarationen
- Performanz
  ...wie `type`-Deklarationen

Somit:

`newtype`-Deklarationen vereinen

- (die besten) Charakteristika von `data`- und `type`-Deklarationen und stellen insofern eine Spezial-/Mischform dar

Es gilt (2)

Aber: Alles hat seinen Preis ("there is no free lunch")!

`newtype`-Deklarationen sind

- auf Typen mit nur einem Konstruktor und einem Feld eingeschränkt
  $\leadsto$ der Preis, Typsicherheit mit Performanz zu verbinden!

Das heißt:

```
newtype Person
  = Pers (Vorname, Nachname, Geschlecht, Alter)
```

ist möglich.

```
newtype Person
  = Pers Vorname Nachname Geschlecht Alter
```

ist jedoch nicht möglich.

type- vs. newtype-Deklarationen (1)

Beispiel: Adressbuch mittels type-Deklaration

```
type Name        = String
type Anschrift   = String
type Adressbuch = [(Name,Anschrift)]

gibAnschrift :: Name -> Adressbuch -> Anschrift
gibAnschrift name ((n,a):r)
   | name == n    = a
   | otherwise    = gibAnschrift name r
gibAnschrift _ [] = error "Anschrift unbekannt"
```

type- vs. newtype-Deklarationen (2)

Beispiel: Adressbuch mittels newtype-Deklaration

```
newtype Name        = N String
newtype Anschrift   = A String
type Adressbuch     = [(Name,Anschrift)]

gibAnschrift :: Name -> Adressbuch -> Anschrift
gibAnschrift (N name) ((N n,a):r)
   | name == n      = a
   | otherwise      = gibAnschrift (N name) r
gibAnschrift _ [] = error "Anschrift unbekannt"
```

`type`- vs. `newtype`-Deklarationen (3)

Das Beispiel zeigt:

- Datenkonstruktoren (wie `N` und `A`) müssen explizit über die Eingabeparameter entfernt werden, um die "eigentlichen" Werte ansprechen zu können
- Werte von mit einer `newtype`-Deklaration definierten Typen können nicht unmittelbar auf der Konsole ausgegeben werden
  - Der (naive) Versuch führt zu einer Fehlermeldung
  - Die Ausgabe solcher Werte wird in Kapitel 8 im Zusammenhang mit der Typklasse `Show` besprochen

Kapitel 6.4.3

Resümee

Resümee (1)

Folgende Faustregeln helfen die Wahl zwischen `type`-, `newtype`- und `data`-Deklarationen zu treffen:

- `type`-Deklarationen führen einen neuen Namen für einen existierenden Typ ein.

  ⇝ `type`-Deklarationen sind deshalb insbesondere sinnvoll, um die Transparenz in Signaturen durch "sprechendere" Typnamen zu erhöhen.
- `newtype`-Deklarationen führen einen neuen Typ für einen existierenden Typ ein.

  ⇝ `newtype`-Deklarationen schaffen deshalb zusätzlich zu sprechenderen Typnamen Typsicherheit ohne Laufzeitstrafkosten und sind darüberhinaus besonders nützlich, um Typen zu Instanzen von Typklassen zu machen (siehe Kapitel 8).

Resümee (2)

- `data`-Deklarationen kreieren neue Typen.

  $\rightsquigarrow$ Neben Typsicherheit und der Möglichkeit, sprechende Typnamen zu wählen, erlauben `data`-Deklarationen völlig neue bisher nicht existierende Datentypen einzuführen.

Kapitel 6.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 6 (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, 2007. (Kapitel 10.1, Type declarations; Kapitel 10.2, Data declarations; Kapitel 10.3, Recursive types)

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 7, Making our own Types and Type Classes; Kapitel 12, Monoids – Wrapping an Existing Type into a New Type)
- 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)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 2, Types and Functions; Kapitel 3, Defining Types, Streamlining Functions)

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 14, Algebraic types)
- Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14, Algebraic types)

Teil III

Funktionale Programmierung

Kapitel 7

Funktionen höherer Ordnung

Überblick

Funktionen höherer Ordnung (kurz: Funktionale)

- Funktionen als Argumente
- Funktionen als Resultate

...der Schritt von applikativer zu funktionaler Programmierung.

Anwendungen:

- Funktionale auf Listen als wichtiger Spezialfall
- Anwendungen auf weiteren Datenstrukturen

Kapitel 7.1

Einführung und Motivation

Funktionale

Funktionen, unter deren Argumenten oder Resultaten Funktionen sind, heißen Funktionen höherer Ordnung oder kurz Funktionale.

Mithin:

Funktionale sind spezielle Funktionen!

Also nichts besonderes, oder?

Funktionale nichts besonderes?

Im Grunde nicht.

Drei kanonische Beispiele aus Mathematik und Informatik:

- Mathematik: *Differential- und Integralrechnung*
  - $\frac{df(x)}{dx}$ $\rightsquigarrow$ `diff f a`
    ...Ableitung von `f` an der Stelle `a`
  - $\int_a^b f(x)dx$ $\rightsquigarrow$ `integral f a b`
    ...Integral von `f` zwischen `a` und `b`

Funktionale nichts besonderes? (fgs.)

- Informatik: *Semantik von Programmiersprachen*
  - Denotationelle Semantik der while-Schleife

    $\mathcal{S}_{ds}[\![\, \texttt{while b do S od} \,]\!] : \Sigma \to \Sigma$

    ...kleinster Fixpunkt eines Funktionals auf der Menge der Zustandstransformationen $[\Sigma \to \Sigma]$, wobei $\Sigma =_{df} \{\sigma \mid \sigma \in [\mathit{Var} \ \to \ \mathit{Data}]\}$ die Menge der (Programm-) Zustände bezeichnet, *Var* die Menge der Programmvariablen und *Data* einen geeigneten Datenbereich.

    (Siehe z.B. VU 185.183 Theoretische Informatik 2)

Andererseits

"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.

Feststellung

Der systematische Umgang mit Funktionen höherer Ordnung als *"first-class citizens"*

- ist charakteristisch für funktionale Programmierung
- hebt funktionale Programmierung von anderen Programmierparadigmen ab
- ist der Schlüssel zu extrem ausdruckskräftigen, eleganten und flexiblen Programmiermethoden

Ein Ausflug in die Philosophie

Der Mensch wird erst durch Arbeit zum Menschen.
Georg W.F. Hegel (27.08.1770-14.11.1831)

Frei nach Hegel:

Funktionale Programmierung wird erst durch Funktionale zu funktionaler Programmierung!

Des Pudels Kern

...bei Funktionalen:

Wiederverwendung!

(ebenso wie bei Funktionsabstraktion und Polymorphie!)

Diese Aspekte wollen wir in der Folge herausarbeiten.

Kapitel 7.2

Funktionale Abstraktion

Abstraktionsprinzipien

Kennzeichnendes Strukturierungsprinzip für

- Prozedurale (und objektorientierte) Sprachen
  - Prozedurale Abstraktion
- Funktionale Sprachen
  - Funktionale Abstraktion
    - 1. Stufe: Funktionen
    - Höherer Stufe: Funktionale

Funktionale Abstraktion (1. Stufe)

Idee:

Sind viele strukturell gleiche Berechnungen auszuführen wie

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

so nimm eine funktionale Abstraktion vor, d.h. schreibe eine Funktion

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)
```

die dann geeignet mit Argumenten aufgerufen wird.

Funktionale Abstraktion (1. Stufe) (fgs.)

Gewinn durch funktionale Abstraktion: Wiederverwendung!

In unserem Beispiel etwa kann jetzt die Berechnungsvorschrift `(a * b + c) * (b + a * c)` wiederverwendet werden:

```
f 5 37 13 ->> 20.196
f 15 7 12 ->> 21.879
f 25 3 10 ->> 21.930
...
```

Eng verwandt zu funktionaler Abstraktion erster Stufe:

- Prozedurale Abstraktion

Funktionale Abstraktion höherer Stufe (1)

(siehe Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach, Addison-Wesley, 1999, S. 7f.)

Betrachte folgende Beispiele:

- 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:

```
natSquSum n | n==0   = 0
            | n>0    = n*n + natSquSum (n-1)
```

Funktionale Abstraktion höherer Stufe (2)

Beobachtung:

- Die Definitionen von `fac`, `natSum` und `natSquSum` folgen demselben Rekursionsschema.

Dieses zugrundeliegende gemeinsame Rekursionsschema ist gekennzeichnet durch:

- Festlegung eines Wertes der Funktion im
  - Basisfall
  - verbleibenden rekursiven Fall als Kombination des Argumentwerts `n` und des Funktionswerts für `n-1`

Funktionale Abstraktion höherer Stufe (3)

Dies legt nahe:

- Obiges Rekursionsschema, gekennzeichnet durch Basisfall und Funktion zur Kombination von Werten, herauszuziehen (zu abstrahieren) und musterhaft zu realisieren.

Wir erhalten als herausgezogenes Rekursionsschema:

```
recScheme base comb n
  | n==0  = base
  | n>0   = comb n (recScheme base comb (n-1))
```

Funktionale Abstraktion höherer Stufe (4)

Funktionale Abstraktion höherer Stufe erlaubt nun

- die Einzelimplementierungen der Rechenvorschriften `fac`, `natSum` und `natSquSum`

zu ersetzen durch

- geeignete Aufrufe des Funktionals `recScheme`.

Damit:

- Wiederverwendung des gemeinsamen Strukturmusters der Funktionen `fac`, `natSum` und `natSquSum`

Funktionale Abstraktion höherer Stufe (5)

Funktionale Abstraktion höherer Stufe:

```
fac       = recScheme 1 (*)

natSum    = recScheme 0 (+)

natSquSum = recScheme 0 (\x y -> x*x + y)
```

In argumentbehafteter Ausführung:

```
fac n       = recScheme 1 (*) n

natSum n    = recScheme 0 (+) n

natSquSum n = recScheme 0 (\x y -> x*x + y) n
```

Funktionale Abstraktion höherer Stufe (6)

Unmittelbarer Vorteil obigen Vorgehens:

- Wiederverwendung und dadurch
  - kürzerer, verlässlicherer, wartungsfreundlicherer Code

Erforderlich für erfolgreiches Gelingen:

- Funktionen höherer Ordnung; kürzer: Funktionale.

  Intuition: Funktionale sind (spezielle) Funktionen, die Funktionen als Argumente erwarten und/oder als Resultat zurückgeben.

Funktionale Abstraktion höherer Stufe (7)

Zusammengefasst am obigen Beispiel:

- Die Untersuchung des Typs von `recScheme`

```
recScheme ::
     Int -> (Int -> Int -> Int) -> Int -> Int
```

  zeigt:
  - `recScheme` ist ein Funktional!

In der Anwendungssituation des Beispiels gilt weiter:

	Wert i. Basisf. (base)	Fkt. z. Kb. v. W. (comb)
`fac`	1	`(*)`
`natSum`	0	`(+)`
`natSquSum`	0	`\x y -> x*x + y`

Bemerkung: Allgemeinster `recScheme`-Typ

Auf Kapitel 8 und 13 vorgreifend gilt für den allgemeinsten Typ von `recScheme`:

```
recScheme :: (Num a, Ord a) =>
                    b -> (a -> b -> b) -> a -> b
```

Fkt. Abstrakt. höh. Stufe am Eingangsbsp. (1)

- Funktionale Abstraktion 1. Stufe führt von Ausdrücken

  ```
  (5*37+13)*(37+5*13), (15*7+12)*(7+15*12),...
  ```

  zu einer Funktion:

  ```
  f :: Int -> Int -> Int -> Int
  f a b c = (a * b + c) * (b + a * c)
  ```

- Funktionale Abstraktion höherer Stufe führt von dieser weiter zu einer Funktion 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
  fho f 25 3 10 ->> 21.930
  ...
  ```

Fkt. Abstrakt. höh. Stufe am Eingangsbsp. (2)

Anders als die Funktion `f` erlaubt die Funktion höherer Ordnung `fho`

- nicht nur die freie Angabe der (elementaren) Argument(-werte),
- sondern auch die freie Angabe ihrer Kombination, d.h. der Verknüpfungs-, Berechnungsvorschrift.

Beispiele:

```
f :: Int -> Int -> Int -> Int
f  a b c = (a * b + c) * (b + a * c)

f2 :: Int -> Int -> Int -> Int
f2 a b c = a^b 'div' c

f3 :: Int -> Int -> Int -> Int
f3 a b c = if (a 'mod' 2 == 0) then b else c
```

Fkt. Abstrakt. höh. Stufe am Eingangsbsp. (3)

Aufrufbeispiele:

```
fho f  2 3 5 ->> f  2 3 4
             ->> (2*3+5)*(3+2*5)
             ->> (6+5)*(3+10)
             ->> 11*13
             ->> 143

fho f2 2 3 5 ->> f2 2 3 5
             ->> 2^3 'div' 5
             ->> 8 'div' 5
             ->> 1

fho f3 2 3 5 ->> f3 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
```

Kapitel 7.3

Funktionen als Argument

Funktionale: Funktionen als Argumente (1)

Anstatt zweier spezialisierter Funktionen

```
max :: Ord a => a -> a -> a
max x y
  | x > y      = x
  | otherwise = y

min :: Ord a => a -> a -> a
min x y
  | x < y      = x
  | otherwise = y
```

Funktionale: Funktionen als Argumente (2)

...eine mit einem Funktions-/Prädikatsargument parametrisierte Funktion:

```
extreme :: Ord a => (a -> a -> Bool) -> a -> a -> a
extreme p m n
  | p m n     = m
  | otherwise = n
```

Anwendungsbeispiele:

```
extreme (>) 17 4 = 17
extreme (<) 17 4 =  4
```

Dies ermöglicht folgende alternative Festlegungen von `max` und `min`:

```
max = extreme (>)  bzw.  max x y = extreme (>) x y
min = extreme (<)  bzw.  min x y = extreme (<) x y
```

Weitere Bsp. für Funktionen als Argumente

Transformation der Marken eines benannten Baums bzw. Herausfiltern der Marken mit einer bestimmten Eigenschaft:

```
data Tree a = Nil | Node a (Tree a) (Tree a)

mapTree :: (a -> a) -> Tree a -> Tree a
mapTree f Nil = Nil
mapTree f (Node elem t1 t2) =
  (Node (f elem)) (mapTree f t1) (mapTree f t2)

filterTree :: (a -> Bool)  -> Tree a -> [a]
filtertree p Nil = []
filterTree p (Node elem t1 t2)
 | p elem    = [elem] ++ (filterTree p t1)
                      ++ (filterTree p t2)
 | otherwise = (filterTree p t1) ++ (filterTree p t2)
```

...mithilfe von Funktionalen, die in Transformationsfunktion bzw. Prädikat parametrisiert sind.

Resümee über Funktionen als Argumente (1)

Funktionen als Argumente

- erhöhen die Ausdruckskraft erheblich und
- unterstützen Wiederverwendung.

Beispiel:

Vergleiche

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip  _      _     = []
```

mit

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f  _      _     = []
```

Resümee über Funktionen als Argumente (2)

Es gilt:

- `zip` lässt sich mithilfe von `zipWith` implementieren
  ⇝ somit: `zipWith` echt genereller als `zip`

```
zip :: [a] -> [b] -> [(a,b)]
zip xs ys  = zipWith h xs ys

h :: a -> b -> (a,b)
h x y = (x,y) -- gleichbedeutend zu:
              -- h x y = (,) x y
```

bzw. mit argumentfreier Variante von `h`:

```
h :: a -> b -> (a,b)
h = (,)
```

Kapitel 7.4

Funktionen als Resultat

Funktionale: Funktionen als Resultate (1)

Auch diese Situation ist bereits aus der Mathematik vertraut:

Etwa in Gestalt der

- Funktionskomposition (Komposition von Funktionen)

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Beispiel:

Theorem (Analysis 1)

Die Komposition stetiger Funktionen ist wieder eine stetige Funktion.

Funktionale: Funktionen als Resultate (2)

...ermöglichen Funktionsdefinitionen auf dem (Abstraktions-) Niveau von Funktionen statt von (elementaren) Werten.

Beispiel:

```
giveFourthElem :: [a] -> a
giveFourthElem = head . tripleTail

tripleTail :: [a] -> [a]
tripleTail = tail . tail . tail
```

Funktionale: Funktionen als Resultate (3)

...sind in komplexen Situationen einfacher zu verstehen und zu ändern als ihre argumentversehenen Gegenstücke

Beispiel:

Vergleiche folgende zwei argumentversehene Varianten der Funktion `giveFourthElem :: [a] -> a`

```
giveFourthElem ls = (head . tripleTail) ls -- Var.1
giveFourthElem ls = head (tripleTail ls)   -- Var.2
```

...mit der argumentlosen Variante

```
giveFourthElem = head . tripleTail
```

Weitere Bsp. für Funktionen als Resultate (1)

Iterierte Funktionsanwendung:

```
iterate :: Int -> (a -> a) -> (a -> a)
iterate n f
  | n > 0     = f . iterate (n-1) f
  | otherwise = id

id :: a -> a            -- Typvariable und
id a = a                -- Argument können
                        -- gleichbenannt sein
```

Aufrufbeispiel:

```
(iterate 3 square) 2
   ->> (square . square . square . id) 2
   ->> 256
```

Weitere Bsp. für Funktionen als Resultate (2)

Vertauschen von Argumenten:

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

Aufrufbeispiel (und Eigenschaft von flip):

```
flip . flip ->> id
```

Weitere Bsp. für Funktionen als Resultate (3)

Anheben (engl. *lifting*) eines Wertes zu einer (konstanten) Funktion:

```
cstFun :: a -> (b -> a)
cstFun c = \x -> c
```

Aufrufbeispiele:

```
cstFun 42 "Die Anwort auf alle Fragen" ->> 42
cstFun iterate giveFourthElem          ->> iterate
(cstFun iterate (+) 3 (\x->x*x)) 2     ->> 256
```

Weitere Bsp. für Funktionen als Resultate (4)

Partielle Auswertung:

Schlüssel: ...partielle Auswertung / partiell ausgewertete Operatoren

- Spezialfall: Operatorabschnitte

- `(*2)` ...die Funktion, die ihr Argument verdoppelt.
- `(2*)` ...s.o.
- `(42<)` ...das Prädikat, das sein Argument daraufhin überprüft, größer 42 zu sein.
- `(42:)` ...die Funktion, die 42 an den Anfang einer typkompatiblen Liste setzt.
- ...

Weitere Bsp. für Funktionen als Resultate (5)

Partiell ausgewertete Operatoren

...besonders elegant und ausdruckskräftig in Kombination mit Funktionalen und Funktionskomposition.

Beispiele:

```
fancySelect :: [Int] -> [Int]
fancySelect = filter (42<) . map (*2)
```

$\leadsto$ multipliziert jedes Element einer Liste mit 2 und entfernt anschließend alle Elemente, die kleiner oder gleich 42 sind.

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

$\leadsto$ kehrt eine Liste um.

Bem.: `map`, `filter` und `foldl` werden in Kürze im Detail besprochen.

Anmerkungen zur Funktionskomposition

Beachte:

Funktionskomposition

- ist assoziativ, d.h.
  `f . (g . h) = (f . g) . h = f . g . h`
- erfordert aufgrund der Bindungsstärke explizite Klammerung. (Bsp.: `head . tripleTail ls` in Variante 1 von Folie "Funktionale: Funktionen als Resultate (3)" führt zu Typfehler.)
- darf auf keinen Fall mit Funktionsapplikation verwechselt werden:
  `f . g` (Komposition)
  ist verschieden von
  `f g` (Applikation)!

Resümee über Funktionen als Resultate

...und Funktionen (gleichberechtigt zu elementaren Werten) als Resultate zuzulassen:

- Ist der Schlüssel, Funktionen miteinander zu verknüpfen und in Programme einzubringen
- Zeichnet funktionale Programmierung signifikant vor anderen Programmierparadigmen aus
- Ist maßgeblich für die Eleganz und Ausdruckskraft und Prägnanz funktionaler Programmierung.

Damit bleibt (möglicherweise) die Schlüsselfrage:

- Wie erhält man funktionale Ergebnisse?

Standardtechniken

... zur Entwicklung von Funktionen mit funktionalen Ergebnissen:

- Explizites Ausprogrammieren
  (Bsp.: `extreme`, `iterate`,...)
- Partielle Auswertung (curryfizierter Funktionen)
  (Bsp.: `curriedAdd 4711 :: Int->Int`,
  `iterate 5 :: (a->a)->(a->a)`,...)
  - Spezialfall: Operatorabschnitte
    (Bsp.: `(*2)`, `(<2)`,...)
- Funktionskomposition
  (Bsp.: `tail . tail . tail :: [a]->[a]`,...)
- λ-Lifting
  (Bsp.: `cstFun :: a -> (b -> a)`,...)

Kapitel 7.1

Funktionale auf Listen

Funktionale auf Listen

...als wichtiger Spezialfall.

Häufige Problemstellungen:

- Transformieren aller Elemente einer Liste in bestimmter Weise
- Herausfiltern aller Elemente einer Liste mit bestimmter Eigenschaft
- Aggregieren aller Elemente einer Liste mittels eines bestimmten Operators
- ...

Funktionale auf Listen

...werden in fkt. Programmiersprachen in großer Zahl für häufige Problemstellungen vordefiniert bereitgestellt, auch in Haskell.

Darunter insbesondere auch Funktionale zum Transformieren, Filtern und Aggregieren von Listen:

- `map` (Transformieren)
- `filter` (Filtern)
- `fold` (genauer: `foldl`, `foldr`) (Aggregieren)

Das Funktional map: Transformieren (1)

Signatur:

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

Variante 1: Implementierung mittels (expliziter) Rekursion:

```
map f []     = []
map f (l:ls) = f l : map f ls
```

Variante 2: 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]
```

Das Funktional `map`: Transformieren (2)

Weitere Anwendungsbeispiele:

```
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]
```

Das Funktional map: Transformieren (3)

Einige Eigenschaften von map:

- Generell 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
```

- (Nur) für strikte f gilt:

```
f . head  =  head . (map f)
```

Das Funktional `filter`: Filtern

Signatur:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Variante 1: Implementierung mittels (expliziter) Rekursion:

```
filter p []       = []
filter p (l:ls)
  | p l         = l : filter p ls
  | otherwise =     filter p ls
```

Variante 2: Implementierung mittels Listenkomprehension:

```
filter p ls = [ l | l <- ls, p l ]
```

Anwendungsbeispiel:

```
filter isPowerOfTwo [2,4..100] = [2,4,8,16,32,64]
```

Aggregieren bzw. Falten von Listen

Aufgabe:

- Berechne die Summe der Elemente einer Liste
  `sum [1,2,3,4,5] ->> 15`

Lösungsidee:

Zwei Rechenweisen sind naheliegend:

- 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 die diesen Rechenweisen zugrundeliegende Idee.

Das Funktional `fold`: Aggregieren (1)

"Falten" von rechts: `foldr`

Signatur ("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)
```

Anwendungsbeispiel:

```
foldr (+) 0 [2,4..10]
     ->> (+ 2 (+ 4 (+ 6 (+ 8 (+ 10 0)))))
     ->> (2 + (4 + (6 + (8 + (10 + 0))))) ->> 30
foldr (+) 0 [] ->> 0
```

In obiger Definition bedeuten: `f`: binäre Funktion, `e`: Startwert, `(l:ls)`: Liste der zu aggregierenden Werte.

Das Funktional `fold`: Aggregieren (2)

Anwendungen von `foldr` zur Definition einiger Standardfunktionen in Haskell:

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

and :: [Bool] -> Bool
and bs = foldr (&&) True bs

sum :: Num a => [a] -> a
sum ls = foldr (+) 0 ls
```

Das Funktional `fold`: Aggregieren (3)

"Falten" von links: `foldl`

Signatur ("zusammenfassen von links"):

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

Mittels (expliziter) Rekursion:

```
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls
```

Anwendungsbeispiel:

```
foldl (+) 0 [2,4..10]
     ->> (+ (+ (+ (+ (+ 0 2) 4) 6) 8) 10)
     ->> (((((0 + 2) + 4) + 6) + 8) + 10) ->> 30
foldl (+) 0 [] ->> 0
```

In obiger Definition bedeuten: `f`: binäre Funktion,
`e`: Startwert, `(l:ls)`: Liste der zu aggregierenden Werte.

Das Funktional `fold`: Aggregieren (4)

`foldr` vs. `foldl` – ein Vergleich:

Signatur ("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))...)
```

Signatur ("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? (1)

Aus Effizienzgründen!

Betrachte und vergleiche:

- concat wie im Prelude mittels foldr definiert:

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

- slowConcat mittels foldl definiert:

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

Warum zwei Faltungsfunktionale? (2)

Unter der Annahme, dass alle Listen `xsi` von gleicher Länge `len` sind, gilt, dass die Kosten der Berechnung von

- ```
  concat [xs1,xs2,...,xsn]
    ->> foldr (++) [] [xs1,xs2,...,xsn]
    ->> xs1 ++ (xs2 ++ (... (xsn ++ [])) ...)
  ```
  durch $n * \texttt{len}$ gegeben sind

- ```
  slowConcat [xs1,xs2,...,xsn]
    ->> foldl (++) [] [xs1,xs2,...,xsn]
    ->> (... (([] ++ xs1) ++ xs2) ...) ++ xsn
  ```
  aber durch
  $\texttt{len} + (\texttt{len} + \texttt{len}) + (\texttt{len} + \texttt{len} + \texttt{len}) + \ldots + (n-1) * \texttt{len}$
  und somit durch $n * (n-1) * \texttt{len}$ gegeben sind

Warum zwei Faltungsfunktionale? (3)

Allgemein gilt:

Es gibt Anwendungsfälle, in denen

- Falten von links
- Falten von rechts (wie bei `concat`)

zu einer effizienteren Berechnung führt (wie auch Anwendungsfälle, in denen kein wesentlicher Unterschied besteht, z.B. für `(+)`).

Die Wahl von `foldr` und `foldl` sollte deshalb stets problem- und kontextabhängig getroffen werden!

Zur Vollständigkeit

Das vordefinierte Funktional `flip`:

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

Anwendungsbeispiel: Listenumkehrung

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

reverse [1,2,3,4,5]  ->>  [5,4,3,2,1]
```

Zur Übung empfohlen: Nachvollziehen, dass `reverse` wie oben definiert die gewünschte Listenumkehrung leistet! Zum Vergleich:

```
reverse []     = []
reverse (l:ls) = (reverse ls) ++ (l:[])
```

Resümee

Für funktionale Programmierung ist typisch:

- Elemente (Werte/Objekte) aller (Daten-) Typen sind Objekte erster Klasse (engl. first-class citizens),

  Informell heißt das:

  Jedes Datenobjekt kann
  - Argument und Wert einer Funktion sein
  - in einer Deklaration benannt sein
  - Teil eines strukturierten Objekts sein

Folgendes Beispiel

...illustriert dies sehr kompakt:

```
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)
```

(Preis-) Fragen:

- Welchen Typ hat `magicType`?
- Wie ist es Hugs möglich, diesen Typ zu bestimmen?

Tipp:

- Hugs fragen: `Main>:t magicType`

Resümee zum Rechnen mit Funktionen

Im wesentlichen sind es folgende Quellen, die Funktionen in Programme einzuführen erlauben:

- Explizite Definition im (Haskell-) Skript
- Ergebnis anderer Funktionen/Funktionsanwendungen
  - Explizit mit funktionalem Ergebnis
  - Partielle Auswertung
    - Spezialfall: Operatorabschnitte
  - Funktionskomposition
  - λ-Lifting

Vorteile der Programmierung mit Funktionalen

- Kürzere und i.a. einfacher zu verstehende Programme
  ...wenn man die Semantik (insbesondere) der grundlegenden Funktionen und Funktionale (`map`, `filter`,...) verinnerlicht hat.
- Einfachere Herleitung und einfacherer Beweis von Programmeigenschaften (Stichwort: Programmverifikation)
  ...da man sich auf die Eigenschaften der zugrundeliegenden Funktionen abstützen kann.
- ...
- Wiederverwendung von Programmcode
  ...und dadurch Unterstützung des Programmierens im Großen.

Stichwort Wiederverwendung

Wesentliche Quellen für Wiederverwendung in funktionalen Programmiersprachen sind:

- Funktionen höherer Ordnung (Kapitel 7)
- Polymorphie (auf Funktionen und Datentypen) (Kapitel 8)

Kapitel 7.6

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 7 (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 on Higher-Order Functions)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 7 (2)

- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (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 7 (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 8

Polymorphie

Polymorphie

Bedeutung lt. Duden:

- Vielgestaltigkeit, Verschiedengestaltigkeit

  ...mit speziellen 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 Wiesen, die Felder, die Tiere)
  - Informatik, speziell Theorie der Programmiersprachen: $\leadsto$ das Thema dieses Kapitels!

Polymorphie

...im programmiersprachlichen Kontext unterscheiden wir insbesondere zwischen:

- Polymorphie auf
  - Funktionen
    - Parametrische Polymorphie (Synonym: "Echte" Polymorphie)
      $\rightsquigarrow$ Sprachmittel: Typvariablen
    - Ad-hoc Polymorphie (Synonyme: "Unechte" Polymorphie, Überladung)
      $\rightsquigarrow$ Haskell-spezifisches Sprachmittel: Typklassen
  - Datentypen
    - Algebraische Datentypen (`data`, `newtype`)
    - Typsynonyme (`type`)

Polymorphe Typen

...ein (Daten-) Typ, Typsynonym `T` heißt polymorph, wenn bei der Deklaration von `T` der Grundtyp oder die Grundtypen der Elemente (in Form einer oder mehrerer Typvariablen) als Parameter angegeben werden.

Beispiele:

```
data Tree a b = Leaf a b
              | Branch (Tree a b) (Tree a b)

data List a = Empty
            | Head a (List a)

newtype Polytype a b c = P (a,b,b,c,c,c)

type Sequence a = [a]
```

Polymorphe Funktionen

...eine Funktion `f` heißt polymorph, wenn deren Parameter (in Form einer oder mehrerer Typvariablen) für Argumente unterschiedlicher Typen definiert sind.

Beispiele:

```
depth :: (Tree a b) -> Int
depth Leaf _ _      = 1
depth (Branch l r) = 1 + max (depth l) (depth r)
```

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

```
lgthList :: List a -> Int
lgthList Empty        = 0
lgthList (Head _ hs) = 1 + lthList hs
```

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

Kapitel 8.1

Polymorphie auf Funktionen

Polymorphie auf Funktionen

Wir unterscheiden:

- Parametrische Polymorphie
- Ad-hoc Polymorphie

Kapitel 8.1.1

Parametrische Polymorphie

Polymorphie

Parametrische Polymorphie auf Funktionen

- haben wir an verschiedenen Beispielen bereits kennengelernt:
  - Die Funktionen `length`, `head` und `tail`
  - Die Funktionale `curry` und `uncurry`
  - Die Funktionale `map`, `filter`, `foldl` und `foldr`
  - ...

Rückblick (1)

Die Funktionen `length`, `head` und `tail`:

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

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

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

Rückblick (2)

Die Funktionale curry und uncurry:

```
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
```

Rückblick (3)

Die Funktionale `map`, `filter`, `foldl` und `foldr`:

```
map :: (a -> b) -> [a] -> [b]
map f ls = [ f l | l <- ls ]

filter :: (a -> Bool) -> [a] -> [a]
filter p ls = [ l | l <- ls, p l ]

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (l:ls) = f l (foldr f e ls)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (l:ls) = foldl f (f e l) ls
```

Kennzeichen parametrischer Polymorphie

Statt

- (ausschließlich) konkreter Typen (wie `Int`, `Bool`, `Char`,...)

treten in der (Typ-) Signatur der Funktionen

- (auch) Typparameter, sog. Typvariablen

auf.

Beispiele:

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

length :: [a] -> Int

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

Typvariablen in Haskell

Typvariablen in Haskell sind:

- freigewählte Identifikatoren, die mit einem Kleinbuchstaben beginnen müssen
  z.B.: `a`, `b`, `fp185A03`,...

Beachte:

Typnamen, (Typ-) Konstruktoren sind im Unterschied dazu in Haskell:

- freigewählte Identifikatoren, die mit einem Großbuchstaben beginnen müssen
  z.B.: `A`, `B`, `String`, `Node`, `FP185A03`,...

Warum Polymorphie auf Funktionen?

Wiederverwendung (durch Abstraktion)!

- wie schon bei Funktionen
- wie schon bei Funktionalen
- ein typisches Vorgehen in der Informatik!

Motivation parametrischer Polymorphie (1)

Listen können Elemente sehr unterschiedlicher Typen zusammenfassen, z.B.

- Listen von Basistypelementen
  `[2,4,23,2,53,4] :: [Int]`
- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [[Int]]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3),(1.12,2.2)] :: [Point]`
- Listen von Bäumen
  `[Nil, Node fac Nil Nil, Node fib (Node (*1000) Nil Nil) Nil] :: [BinTree1]`
- Listen von Funktionen
  `[fun91, fib, (+1), (*2)] :: [Integer -> Integer]`
- ...

Motivation parametrischer Polymorphie (2)

- Aufgabe:
  - Bestimme die Länge einer Liste, d.h. die Anzahl ihrer Elemente.
- Naive Lösung:
  - Schreibe für jeden Typ eine entsprechende Funktion.

Motivation parametrischer Polymorphie (3)

Umsetzung der naiven Lösung:

```
lengthIntLst :: [Int] -> Int
lengthIntLst []     = 0
lengthIntLst (_:xs) = 1 + lengthIntLst xs

lengthIntLstLst :: [[Int]] -> Int
lengthIntLstLst []     = 0
lengthIntLstLst (_:xs) = 1 + lengthIntLstLst xs

lengthPointLst :: [Point] -> Int
lengthPointLst []     = 0
lengthPointLst (_:xs) = 1 + lengthPointLst xs

lengthTreeLst :: [BinTree1] -> Int
lengthTreeLst []     = 0
lengthTreeLst (_:xs) = 1 + lengthTreeLst xs

lengthFunLst :: [Integer -> Integer] -> Int
lengthFunLst []     = 0
lengthFunLst (_:xs) = 1 + lengthFunLst xs
```

Motivation parametrischer Polymorphie (4)

Die vorigen Deklarationen erlauben z.B. folgende Aufrufe:

```
lengthIntLst [2,4,23,2,53,4] ->> 6

lengthIntLstLst [[2,4,23,2,5],[3,4],[],[56,7,6,]] ->> 4

lengthPointLst [(3.14,42.0),(56.1,8.3),(1.2,2.2)] ->> 3

lengthTreeLst [Nil, Node 42 Nil Nil,
               Node 17 (Node 4 Nil Nil) Nil)]     ->> 3

lengthTreeLst
      [Nil, Node fac Nil Nil,
       Node fib (Node (*1000) Nil Nil) Nil)]      ->> 3

lengthFunLst [fac, fib, fun91, (+1), (*2)]        ->> 5
```

Motivation parametrischer Polymorphie (5)

Beobachtung:

- Die einzelnen Rechenvorschriften zur Längenberechnung sind i.w. identisch
- Unterschiede beschränken sich auf
  - Funktionsnamen und
  - Typsignaturen

Motivation parametrischer Polymorphie (6)

Sprachen, die parametrische Polymorphie offerieren, erlauben eine elegantere Lösung unserer Aufgabe:

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

length [2,4,23,2,53,4] ->> 6
length [[2,4,23,2,5],[3,4],[],[56,7,6,]] ->> 4
length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] ->> 3
length [Nil, Node 42 Nil Nil,
        Node 17 (Node 4 Nil Nil) Nil)] ->> 3
length [Nil, Node fac Nil Nil,
        Node fib (Node (*1000) Nil Nil) Nil)] ->> 3
length [fac, fib, fun91, (+1), (*2)] ->> 5
```

Funktionale Sprachen, auch Haskell, offerieren parametrische Polymorphie!

Motivation parametrischer Polymorphie (7)

Unmittelbare Vorteile parametrischer Polymorphie:

- Wiederverwendung von
  - Verknüpfungs-, Auswertungsvorschriften
  - Funktionsnamen (*Gute Namen sind knapp!*)

Polymorphie vs. Monomorphie

- Polymorphie:
  Rechenvorschriften der Form
  - `length :: [a] -> Int`

  heißen polymorph.

- Monomorphie:
  Rechenvorschriften der Form
  - `lengthIntLst :: [Int] -> Int`
  - `lengthIntLstLst :: [[Int]] -> Int`
  - `lengthPointLst :: [Point] -> Int`
  - `lengthFunLst :: [Integer -> Integer] -> Int`
  - `lengthTreeLst :: [BinTree1] -> Int`

  heißen monomorph.

Sprechweisen im Zshg. m. param. Polymorphie

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

Bezeichnungen:

- `a` in der Typsignatur von `length` heißt Typvariable.
  Typvariablen werden mit Kleinbuchstaben gewöhnlich vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...
- Typen der Form

  ```
  length :: [Point] -> Int
  length :: [[Int]] -> Int
  length :: [Integer -> Integer] -> Int
  ...
  ```

  heißen Instanzen des Typs `[a] -> Int`. Letzterer heißt allgemeinster Typ der Funktion `length`.

Anmerkung

Das Hugs-Kommando `:t` liefert stets den (eindeutig bestimmten) allgemeinsten Typ eines (wohlgeformten) Haskell-Ausdrucks `expr`.

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)
```

Weitere Beispiele polymorpher Funktionen (1)

Identitätsfunktion:

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

id 3 ->> 3
id ["abc","def"] ->> ["abc","def"]
```

Weitere Beispiele polymorpher Funktionen (2)

Reißverschlussfunktion: "Verpaaren" von Listen

```
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))]
```

Weitere Beispiele polymorpher Funktionen (3)

Reißverschlussfunktion: “Entpaaren” von Listen

```
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 in Haskell auf Listen vordefinierte parametrisch polymorphe Funktionen

`:`	`::`	`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]`	Umdrehen einer Liste
...			

Kapitel 8.1.2

Ad-hoc Polymorphie

Ad-hoc Polymorphie

Ad-hoc Polymorphie

- ist eine schwächere, weniger allgemeine Form parametrischer Polymorphie

Synonyme zu ad-hoc Polymorphie sind

- Überladen (engl. Overloading)
- "Unechte" Polymorphie

Motivation von Ad-hoc Polymorphie (1)

Ausdrücke der Form

```
(+) 2 3          ->> 5
(+) 27.55 12.8   ->> 39.63
(+) 12.42 3      ->> 15.42
```

sind Beispiele wohlgeformter Haskell-Ausdrücke; dahingegen sind Ausdrücke der Form

```
(+) True False
(+) 'a' 'b'
(+) [1,2,3] [4,5,6]
```

Beispiele nicht wohlgeformter Haskell-Ausdrücke.

Motivation von Ad-hoc Polymorphie (2)

Offenbar:

- ist (+) nicht monomorph
  ...da (+) für mehr als einen Argumenttyp arbeitet
- ist der Typ von (+) nicht echt polymorph und somit verschieden von `a -> a -> a`
  ...da (+) nicht für jeden Argumenttyp arbeitet

Tatsächlich:

- ist (+) typisches Beispiel eines überladenen Operators.

Das Kommando `:t (+)` in Hugs liefert:

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

Typklassen in Haskell

Informell:

- Eine Typklasse ist eine Kollektion von Typen, auf denen eine in der Typklasse festgelegte Menge von Funktionen definiert ist.
- Die Typklasse `Num` ist die Kollektion der numerischen Typen `Int`, `Integer`, `Float`, etc., auf denen die Funktionen `(+)`, `(*)`, `(-)`, etc. definiert sind.

Zur Übung empfohlen: Vergleiche dieses Klassenkonzept z.B. mit dem Schnittstellenkonzept aus Java. Welche Gemeinsamkeiten und Unterschiede gibt es?

Polymorphie vs. Ad-hoc Polymorphie

Informell:

- (Parametrische) Polymorphie
  $\rightsquigarrow$ gleicher Code trotz unterschiedlicher Typen
- Ad-hoc Polymorphie (synonym: Überladen)
  $\rightsquigarrow$ unterschiedlicher Code trotz gleichen Namens (mit i.a. sinnvollerweise vergleichbarer Funktionalität)

Ein Beispiel zu Typklassen (1)

Wir nehmen an, wir seien an der Größe interessiert von

- Listen und
- Bäumen

Der Begriff "Größe" sei dabei typabhängig, z.B.

- Anzahl der Elemente bei Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...

  bei Bäumen

Ein Beispiel zu Typklassen (2)

Wunsch:

- Wir möchen eine Funktion `size` haben, die mit Argumenten der verschiedenen Typen aufgerufen werden kann und typentsprechend die Größe liefert.

Lösung:

- Ad-hoc Polymorphie und Typklassen

Ein Beispiel zu Typklassen (3)

Wir betrachten folgende Baum- und Listenvarianten:

- Baumvarianten

```
data Tree1 a = Nil
             | Node1 a (Tree1 a) (Tree1 a)

data Tree2 a b
       = Leaf2 b
         | Node2 a b (Tree2 a b) (Tree2 a b)

data Tree3 = Leaf3 String
           | Node3 String Tree3 Tree3
```

- Listenvarianten

```
type Lst a  = [a]

data List a = Empty
            | Head a (List a)
```

Ein Beispiel zu Typklassen (4)

Naive Lösung:

- Schreibe für jeden Typ eine passende Funktion

```
sizeT1 :: Tree1 a -> Int     -- Zählen der Knoten
sizeT1 Nil            = 0
sizeT1 (Node1 _ l r) = 1 + sizeT1 l + sizeT1 r

sizeT2 :: (Tree2 a b) -> Int -- Zählen der
sizeT2 (Leaf2 _)        = 1  -- Benennungen
sizeT2 (Node2 _ _ l r) = 2 + sizeT2 l + sizeT2 r

sizeT3 :: Tree3 -> Int      -- Addieren der Längen
sizeT3 (Leaf3 m)      = length m  -- der Benennungen
sizeT3 (Node3 m l r) = length m + sizeT3 l + sizeT3 r
```

Ein Beispiel zu Typklassen (5)

```
sizeLst :: [a] -> Int        -- Zählen der Elemente
sizeLst = length

sizeList :: (List a) -> Int -- Zählen der Elemente
sizeList Empty         = 0
sizeList (Head _ ls) = 1 + sizeList ls
```

Ein Beispiel zu Typklassen (6)

Lösung mittels unechter Polymorphie und Typklassen:

```
class Size a where            -- Def. der Typklasse Size
  size :: a -> Int

instance Size (Tree1 a) where     -- Instanzbildung
  size Nil              = 0       -- für (Tree1 a)
  size (Node1 n l r) = 1 + size l + size r

instance Size (Tree2 a b) where -- Instanzbildung
  size (Leaf2 _)        = 1       -- für (Tree2 a b)
  size (Node2 _ _ l r) = 2 + size l + size r

instance Size Tree3 where          -- Instanzbildung
  size (Leaf3 m)     = length m -- für Tree3
  size (Node3 m l r) = length m + size l + size r
```

Ein Beispiel zu Typklassen (7)

```
instance Size [a] where                 -- Instanzbildung
  size = length                         -- für Listen

instance Size (List a) where            -- Instanzbildung
  size Empty         = 0                -- für eigendefi-
  size (Head _ ls) = 1 + size ls        -- nierte Listen
```

Ein Beispiel zu Typklassen (8)

Das Kommando `:t size` liefert:

```
size :: Size a => a -> Int
```

Wir erhalten wie gewünscht:

```
size Nil ->> 0
size (Node1 "asdf" (Node1 "jk" Nil Nil) Nil) ->> 2

size (Leaf2 "adf") ->> 1
size ((Node2 "asdf" 3
          (Node2 "jk" 2 (Leaf2 17) (Leaf2 4))
          (Leaf2 21)) ->> 7

size (Leaf3 "abc")    ->> 3
size (Node3 "asdf"
       (Node3 "jkertt" (Leaf3 "abc") (Leaf3 "ac"))
       (Leaf3 "xy"))  ->> 17
```

Ein Beispiel zu Typklassen (9)

```
size [5,3,45,676,7]     ->> 5
size [True,False,True] ->> 3

size Empty                              ->> 0
size (Head 2 (Head 3 Empty))            ->> 2
size (Head 2 (Head 3 (Head 5 Empty))) ->> 3
```

Zusammenfassung

...zur Typklasse `Size` und Funktion `size`:

- die Typklasse `Size` stellt die Typspezifikation der Funktion `size` zur Verfügung
- jede Instanz der Typklasse `Size` muss eine instanzspezifische Implementierung der Funktion `size` zur Verfügung stellen
- Im Ergebnis ist die Funktion `size` wie auch z.B. die in Haskell vordefinierten Operatoren `(+)`, `(*)`, `(-)`, etc., oder die Relatoren `(==)`, `(>)`, `(>=)`, etc. überladen
- Synonyme für Überladen sind ad-hoc Polymorphie und unechte Polymorphie

Polymorphie vs. Ad-hoc Polymorphie

Intuitiv:

- Polymorphie
  Der polymorphe Typ `(a -> a)` wie in der Funktion
  `id :: a -> a` steht abkürzend für:
  $\forall$`(a) a -> a` "für alle Typen"
- Ad-hoc Polymorphie
  Der Typ `(Num a => a -> a -> a)` wie in der Funktion
  `(+) :: Num a => a -> a -> a` steht abkürzend für:
  $\forall$`(a` $\in$ `Num) a -> a -> a` "für alle Typen aus `Num`"

Im Haskell-Jargon ist `Num` eine sog.

- Typklasse

...eine von vielen in Haskell vordefinierten Typklassen.

Exkurs: Nicht alles ist unterstützt (1)

...sei "Größe" für Tupellisten nicht durch die Anzahl der Listenelemente, sondern durch die Anzahl der Komponenten der tupelförmigen Listenelemente bestimmt.

Lösung durch entsprechende Instanzbildung:

```
instance Size [(a,b)] where
  size = (*2) . length

instance Size [(a,b,c)] where
  size = (*3) . length
```

Beachte: Die Instanzbildung `instance Size [(a,b)]` geht über den Standard von Haskell 98 hinaus und ist nur in entsprechenden Erweiterungen möglich.

Exkurs: Nicht alles ist unterstützt (2)

Wie bisher gilt für den Typ der Funktion `size`:

```
size :: Size a => a -> Int
```

Wir erhalten wie erwartet und gewünscht:

```
size [(5,"Smith"),(4,"Hall"),(7,"Douglas")]
                                 ->> 6
size [(5,"Smith",45),(4,"Hall",37),
      (7,"Douglas",42)] ->> 9
```

Exkurs: Nicht alles ist unterstützt (3)

Wermutstropfen:

Die Instanzbildungen

```
instance Size [a] where
  size = length

instance Size [(a,b)] where
  size = (*2) . length

instance Size [(a,b,c)] where
  size = (*3) . length
```

sind nicht gleichzeitig möglich.

Exkurs: Nicht alles ist unterstützt (4)

Problem: Überlappende Typen!

```
ERROR "test.hs:45" - Overlapping instances
                     for class "Size"
*** This instance   : Size [(a,b)]
*** Overlaps with   : Size [a]
*** Common instance : Size [(a,b)]
```

Konsequenz:

- Für Argumente von Instanzen des Typs `[(a,b)]` (und ebenso des Typs `[(a,b,c)]`) ist die Überladung des Operators `size` nicht mehr auflösbar
- Wünschenswert wäre:

```
instance Size [a] w/out [(b,c)],[(b,c,d)] where
  size = length
```

Beachte: Dies ist in dieser Weise in Haskell nicht möglich.

Definition von Typklassen

Allgemeines Muster einer Typklassendefinition:

```
class Name tv where
   ...signature involving the type variable tv
```

wobei

- `Name`: Identifikator der Klasse
- `tv`: Typvariable
- `signature`: Liste von Namen zusammen mit ihren Typen

Zusammenfassung zu Typklassen

Intuitiv:

- Typklassen sind Kollektionen von Typen, für die eine gewisse Menge von Funktionen ("vergleichbarer" Funktionalität) definiert ist.

Beachte:

- "Vergleichbare" Funktionalität kann nicht syntaktisch erzwungen werden; sie liegt in der Verantwortung des Programmierers!

  $\rightsquigarrow$ Appell an die Programmierdisziplin

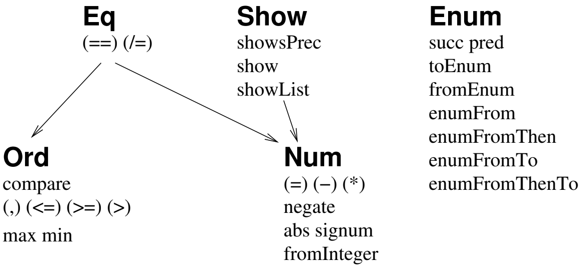
Auswahl in Haskell vordef. Typklassen (1)

Vordefinierte Typklassen in Haskell:

- Gleichheit `Eq`: die Klasse der Typen mit Gleichheitstest und Ungleichheitstest
- Ordnungen `Ord`: die Klasse der Typen mit Ordnungsrelationen (wie $<$, $\leq$, $>$, $\geq$, etc.)
- Aufzählung `Enum`: die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29]`)
- Werte zu Zeichenreichen `Show`: die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können
- Zeichenreihen zu Werten `Read`: die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind
- ...

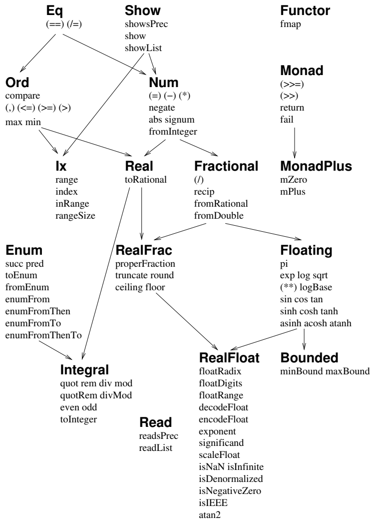
Ausgewahl in Haskell vordef. Typklassen (2)

Auswahl vordefinierter Typklassen, ihrer Abhängigkeiten, Operatoren und Funktionen in "Standard Prelude" nebst Bibliotheken:



Quelle: Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach. Addison-Wesley, 1999, Figure 2.4 (Ausschnitt).

Auswahl in Haskell vordef. Typklassen (3)



Quelle: Fethi Rabhi, Guy Lapalme. Algorithms - A Functional Approach. Addison-Wesley, 1999, Figure 2.4.

Beispiel: Die Typklasse Eq (1)

Die in Haskell vordefinierte Typklasse Eq:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)
```

Die Typklasse Eq stellt

- Typspezifikationen von zwei Wahrheitswertfunktionen
- zusammen mit je einer Protoimplementierung

bereit.

Beispiel: Die Typklasse Eq (2)

Beachte:

- die Protoimplementierungen sind für sich allein nicht ausreichend, sondern stützen sich wechselweise aufeinander ab.

Trotz dieser Unvollständigkeit ergibt sich als Vorteil:

- Bei Instanzbildungen reicht es, entweder eine Implementierung für `(==)` oder für `(/=)` anzugeben. Für den jeweils anderen Operator gilt dann die vordefinierte Proto- (default) Implementierung.
- Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Protoimplementierungen überschrieben.

Instanzbildungen der Typklasse Eq (1)

Am Beispiel des Typs der Wahrheitswerte:

```
instance Eq Bool where
  (==) True True   = True
  (==) False False = True
  (==) _ _         = False
```

Beachte: Der Ausdruck “Instanz” im Haskell-Jargon ist überladen!

- Bislang: Typ `T` ist Instanz eines Typs `U` (z.B. Typ `[Int]` ist Instanz des Typs `[a]`)
- Zusätzlich jetzt: Typ `T` ist Instanz einer (Typ-) Klasse `C` (z.B. Typ `Bool` ist Instanz der Typklasse `Eq`)

Instanzbildungen der Typklasse Eq (2)

Am Beispiel eines Typs für Punkte in der (x,y)-Ebene:

```
newtype Point = Point (Int,Int)

instance Eq Point where
 (==) (Point (x,y)) (Point (u,v))
                    = ((x==u) && (y==v))
```

Erinnerung: Typ- und Konstruktorname (`Point`!) dürfen übereinstimmen.

Instanzbildungen der Typklasse Eq (3)

Auch selbstdefinierte Typen können zu Instanzen vordefinierter Typklassen gemacht werden, z.B. der Baumtyp `Tree1`:

```
data Tree1 = Nil
           | Node1 Int Tree1 Tree1

instance Eq Tree1 where
  (==) Nil Nil                  = True
  (==) (Node1 m t1 t2) (Node1 n u1 u2)
                                = (m  == n)  &&
                                  (t1 == u1) &&
                                  (t2 == u2)
  (==) _ _                      = False
```

Instanzbildungen der Typklasse Eq (4)

Das Vorgenannte gilt in gleicher Weise für selbstdefinierte polymorphe Typen:

```
data Tree2 a   = Leaf2 a
                 | Node2 a (Tree2 a) (Tree2 a)

data Tree3 a b
           = Leaf3 b
             | Node3 a b (Tree3 a b) (Tree3 a b)
```

Instanzbildungen der Typklasse Eq (5)

```
instance (Eq a) => Eq (Tree2 a) where
  (==) (Leaf2 s) (Leaf2 t)                   = (s == t)
  (==) (Node2 s t1 t1) (Node2 t u1 u2) = (s == t)   &&
                                          (t1 == u1) &&
                                          (t2 == u2)
  (==) _ _                                   = False

instance (Eq a, Eq b) => Eq (Tree3 a b) where
  (==) (Leaf3 q) (Leaf3 s)                   = (q == s)
  (==) (Node3 p q t1 t1) (Node3 r s u1 u2)
                                             = (p == r)   &&
                                               (q == s)   &&
                                               (t1 == u1) &&
                                               (t2 == u2)
  (==) _ _                                   = False
```

Instanzbildungen der Typklasse Eq (6)

Instanzbildungen sind flexibel:

Abweichend von der vorher definierten Gleichheitsrelation auf Bäumen vom Typ (Tree3 a b), hätten wir den Gleichheitstest auch so festlegen können, dass die Markierungen vom Typ a in inneren Knoten für den Gleichheitstest irrelevant sind:

```
instance (Eq b) => Eq (Tree3 a b) where
  (==) (Leaf3 q) (Leaf3 s)                   = (q == s)
  (==) (Node3 _ q t1 t1) (Node3 _ s u1 u2)
                                             = (q == s)   &&
                                               (t1 == u1) &&
                                               (t2 == u2)
  (==) _ _                                   = False
```

Beachte, dass für Instanzen des Typs a jetzt nicht mehr Mitgliedschaft in der Typklasse Eq gefordert werden muss.

Bemerkungen

- Getrennt durch Beistriche wie in `(Eq a, Eq b)` können in Kontexten mehrfache — konjunktiv zu verstehende — (Typ-) Bedingungen angegeben werden.
- Damit die Anwendbarkeit des Relators `(==)` auf Werte von Knotenbenennungen gewährleistet ist, müssen die Instanzen der Typvariablen `a` und `b` selbst schon als Instanzen der Typklasse `Eq` vorausgesetzt sein.

Vereinbarungen und Sprechweisen

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t)                = (s == t)
  (==) (Node1 s t1 t1) (Node1 t u1 u2) = (s == t)   &&
                                          (t1 == u1) &&
                                          (t2 == u2)
  (==) _ _                                = False
```

Vereinbarungen und Sprechweisen:

- `Tree1 a` ist Instanz der (gehört zur) Typklasse `Eq`, wenn `a` zu dieser Klasse gehört.
- Der Teil links von `=>` heißt Kontext.
- Rechts von `=>` dürfen ausschließlich Basistypen (z.B. `Int`), Typkonstruktoren beinhaltende Typen (z.B. `Tree a`, `[...]`) oder auf ausgezeichnete Typvariablen angewandte Tupeltypen (z.B. `(a,b,c,d)`) stehen.

Zusammenfassung zur Typklasse Eq

Der Vergleichsoperator (==) der Typklasse Eq ist

- überladen (synonym: ad-hoc polymorph, unecht polymorph), nicht parametrisch polymorph
- in Haskell als Operation in der Typklasse Eq vorgegeben.
- damit anwendbar auf Werte aller Typen, die Instanzen von Eq sind
- viele Typen sind bereits vordefinierte Instanz von Eq, z.B. alle elementaren Typen, Tupel und Listen über elementaren Typen
- auch selbstdefinierte Typen können zu Instanzen von Eq gemacht werden

Frage

- Ist es denkbar, jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?

- De facto hieße das, den Typ des Vergleichsoperators `(==)` von

  ```
  (==) :: Eq a => a -> a -> Bool
  ```

  auf

  ```
  (==) :: a -> a -> Bool
  ```

  zu verallgemeinern.

Nein!

Der Grund ist im Kern folgender:

Anders als z.B. die Länge einer Liste, die eine vom konkreten Listenelementtyp unabhängige Eigenschaft ist und deshalb eine (echt) polymorphe Eigenschaft ist und eine entsprechende Implementierung erlaubt

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

ist Gleichheit eine typabhängige Eigenschaft, die eine typspezifische Implementierung verlangt.

Beispiel:

- Unsere typspezifischen Implementierungen des Gleichheitstests auf Bäumen

Warum ist nicht mehr möglich? (1)

Im Sinne von Funktionen als first class citizens wäre ein Gleichheitstest auf Funktionen höchst wünschenswert.

Zum Beispiel:

```
(==) fac fib                          ->>  False
(==) (\x -> x+x) (\x -> 2*x)  ->>  True
(==) (+2) (2+)                        ->>  True
```

Warum ist nicht mehr möglich? (2)

In Haskell erforderte eine Umsetzung Instanzbildungen der Art:

```
instance Eq (Int -> Int) where
(==) f g = ...

instance Eq (Int -> Int -> Int) where
(==) f g = ...
```

Können wir die “Punkte” so ersetzen, dass wir einen Gleichheitstest für alle Funktionen der Typen `(Int -> Int)` und `(Int -> Int -> Int)` erhalten?

Nein!

Warum ist nicht mehr möglich? (3)

Zwar lässt sich für konkret vorgelegte Funktionen Gleichheit fallweise (algorithmisch) entscheiden, generell aber gilt folgendes aus der Theoretischen Informatik bekannte negative Resultat:

Theorem (Theoretische Informatik)

Gleichheit von Funktionen ist nicht entscheidbar.

Warum ist nicht mehr möglich? (4)

Erinnerung:

“Gleichheit von Funktionen ist nicht entscheidbar” heißt:

- Es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Zur Übung: Machen Sie sich klar, dass daraus nicht folgt, dass Gleichheit zweier Funktionen nie (in endlicher Zeit algorithmisch) entschieden werden kann.

Schlussfolgerungen

...anhand der Beobachtungen am Gleichheitstest (==):

- Offenbar können Funktionen bestimmter Funktionalität nicht für jeden Typ angegeben werden; insbesondere lässt sich nicht für jeden Typ eine Implementierung des Gleichheitsrelators (==) angeben, sondern nur für eine Teilmenge aller möglichen Typen.
- Die Teilmenge der Typen, für die das für den Gleichheitsrelator möglich ist, bzw. eine Teilmenge davon, für die das in einem konkreten Haskell-Programm tatsächlich gemacht wird, ist im Haskell-Jargon eine Kollektion (engl. collection) von Typen, eine sog. Typklasse.

Zusammenfassung

- Auch wenn es verlockend wäre, eine (echt) polymorphe Implementierung von (==) zu haben mit Signatur

  ```
  (==) :: a -> a -> Bool
  ```

  und damit analog zur Funktion zur Längenbestimmung von Listen

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

  ist eine Implementierung in dieser Allgemeinheit für (==) in keiner (!) Sprache möglich!

- Typen, für die eine Implementierung von (==) angegeben werden kann, werden in Haskell in der Typklasse Eq zusammengefasst.

Mehr zu Typklassen: Erben, vererben und überschreiben

Typklassen können (anders als die Typklasse `Size`)

- 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 an ausgewählten Beispielen von in Haskell vordefinierten Typklassen.

Typklassen: (Ver-)erben, überschreiben (1)

Vererbung auf Typklassenebene:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare              :: a -> a -> Ordering
  x <= y         = (x<y) || (x==y)
  x > y          = y < x
  ...
  compare x y
    | x == y     = EQ
    | x <= y     = LT
    | otherwise  = GT
```

Typklassen: (Ver-)erben, überschreiben (2)

- Die (wie `Eq` vordefinierte) Typklasse `Ord` erweitert die Klasse `Eq`.
- Jede Instanz der Typklasse `Ord` muss Implementierungen für alle Funktionen der Klassen `Eq` und `Ord` bereitstellen.

Beachte:

- `Ord` stellt wie `Eq` für einige Funktionen bereits Protoimplementierungen bereit.
- Bei der Instanzbildung für weitere Typen reicht es deshalb, Implementierungen der Relatoren `(==)` und `(<)` anzugeben.
- Durch Angabe instanzspezifischer Implementierungen bei der Instanzbildung können diese Protoimplementierungen aber auch nach Wunsch überschrieben werden.

Typklassen: (Ver-)erben, überschreiben (3)

Auch Mehrfachvererbung auf Typklassenebene ist 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
fromInt       = ...
```

Übung: Vergleiche dies mit Vererbungskonzepten objektorientierter Sprachen!

Typklassen: (Ver-)erben, überschreiben (4)

Überschreiben ererbter Funktionen am Beispiel der Instanz `Point` der Typklasse `Eq`:

- Vererbung:
  Für die Instanzdeklaration von `Point` zur Klasse `Eq`
  ```
  instance Eq Point where
   Point (x,y) == Point (w,z) = (x==w) && (y==z)
  ```
  erbt `Point` folgende Implementierung von `(/=)` aus `Eq`:
  ```
  Point x /= Point y = not (Point x == Point y)
  ```
- Überschreiben:
  Die ererbte (Standard-) Implementierung von `(/=)` kann überschrieben werden, z.B. wie unten durch eine (geringfügig) effizientere Variante:
  ```
  instance Eq Point where
   Point (x,y) == Point (w,z) = (x==w) && (y==z)
   Point x /= Point y = if x/=w then True else y/=z
  ```

Automatische Instanzbildung (1)

(Automatisch) abgeleitete Instanzen von Typklassen:

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                  deriving (Eq,Ord,Enum,Bounded,
                            Show,Read)

data Tree a = Nil
              | Node a (Tree a) (Tree a)
                                 deriving (Eq,Ord)
```

- Algebraische Typen können durch Angabe einer `deriving`-Klausel als Instanzen vordefinierter Klassen automatisch angelegt werden.
- Intuitiv ersetzt die Angabe der `deriving`-Klausel die Angabe einer `instance`-Klausel.

Automatische Instanzbildung (2)

Beispiel: Die Deklaration mit deriving-Klausel

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)
                                  deriving Eq
```

ist gleichbedeutend zu:

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
  (==) Nil Nil                = True
  (==) (Node m t1 t2) (Node n u1 u2)
                              = (m == n)   &&
                                (t1 == u1) &&
                                (t2 == u2)
  (==) _ _                    = False
```

Automatische Instanzbildung (3)

Entsprechend ist die Deklaration

```
data Tree3 a b
 = Leaf3 bl
   | Node3 a b (Tree3 a b) (Tree3 a b) deriving Eq
```

gleichbedeutend zu:

```
data Tree3 a b = Leaf3 bl
                 | Node3 a b (Tree3 a b) (Tree3 a b)

instance (Eq a, Eq b) => Eq (Tree3 a b) where
  (==) (Leaf3 q) (Leaf3 s)  = (q == s)
  (==) (Node3 p q t1 t1) (Node3 r s u1 u2)
                            = (p == r)   &&
                              (q == s)   &&
                              (t1 == u1) &&
                              (t2 == u2)
  (==) _ _                  = False
```

Automatische Instanzbildung (4)

Soll Gleichheit hingegen "unkonventionell" realisiert sein wie in

```
data Tree3 a b = Leaf3 bl
               | Node3 a b (Tree3 a b) (Tree3 a b)

instance (Eq a, Eq b) => Eq (Tree3 a b) where
 (==) (Leaf3 q) (Leaf3 s)          = (q == s)
 (==) (Node3 _ q t1 t1) (Node3 _ s u1 u2)
                                  = (q == s)   &&
                                    (t1 == u1) &&
                                    (t2 == u2)
 (==) _ _                          = False
```

...ist eine explizite Instanzdeklaration erforderlich.

Automatische Instanzbildung (5)

Beachte:

Automatische Instanzbildung ist nicht für beliebige Typklassen möglich, sondern eingeschränkt für die Typklassen

- `Eq`
- `Ord`
- `Enum`
- `Bounded`
- `Show`
- `Read`

Resümee

Parametrische Polymorphie und Überladen auf Funktionen bedeuten:

- vordergründig
  ...ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.
- präziser und tiefgründiger
  - Parametrisch polymorphe Funktionen
    - ...haben eine einzige Implementierung, die für alle (zugelassenen/abgedeckten) Typen arbeitet (Bsp.: `length :: [a] -> Int]`)
  - Überladene Funktionen
    - ...arbeiten für Instanzen einer Klasse von Typen mit einer für jede Instanz spezifischen Implementierung (Bsp.: `size :: Size a => a -> Int`)

Resümee (fgs.)

Vorteile durch parametrische Polymorphie und Überladen:

- Ohne parametrische Polymorphie und Überladen ginge es nicht ohne ausgezeichnete Namen für alle Funktionen und Operatoren.
- Das gälte auch für die bekannten arithmetischen Operatoren; so wären insbesondere Namen der Art $+_{Int}$, $+_{Float}$, $*_{Int}$, $*_{Float}$, etc. erforderlich.
- Deren zwangweiser Gebrauch wäre nicht nur ungewohnt und unschön, sondern in der täglichen Praxis auch lästig.
- Haskells Angebot, hier Abhilfe zu schaffen, sind parametrische Polymorphie und Überladen von Funktionsnamen und Operatoren; wichig für letzteres ist das Konzept der Typklassen in Haskell.

Anmerkung: Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg und bieten andere Konzepte.

Kapitel 8.2

Polymorphie auf Datentypen

Polymorphie

Nach echter und unechter Polymorphie auf Funktionen jetzt

- Polymorphie auf Datentypen
  - Algebraische Datentypen (`data`, `newtype`)
  - Typsynonymen (`type`)

Warum Polymorphie auf Datentypen?

Wiederverwendung (durch Abstraktion)!

- wie schon bei Funktionen
- wie schon bei Funktionalen
- wie schon bei Polymorphie auf Funktionen
- ein typisches Vorgehen in der Informatik!

Beispiel

Ähnlich wie auf Funktionen, hier für curry,

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

...lässt sich allgemein auf algebraischen Typen Typunabhängigkeit vorteilhaft ausnutzen; siehe etwa die Funktion depth:

```
depth :: Tree a -> Int
depth Nil = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

depth (Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil))
                                                  ->> 2
depth (Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil))
                                                  ->> 2
depth (Node "ab" (Node "" Nil Nil) (Node "xyz" Nil Nil))
                                                  ->> 2
```

Polymorphe algebraische Typen (1)

Der Schlüssel:

- Deklarationen algebraischer Typen dürfen Typvariablen enthalten und werden dadurch polymorph

Beispiele:

```
data Pairs a = Pair a a

data Tree a = Nil | Node a (Tree a) (Tree a)

newtype Ptype a b c = P (a,(b,b),[c],(a->b->c))
```

Polymorphe algebraische Typen (2)

Beispiele konkreter Instanzen und Werte:

```
data Pairs a = Pair a a
p1 = Pair 17 4    :: Pairs Int
p2 = Pair [] [42] :: Pairs [Int]
p3 = Pair [] []   :: Pairs [a]
```

```
data Tree a = Nil | Node a (Tree a) (Tree a)
t1 = Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil)
                                 :: Tree Char
t2 = Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)
                                 :: Tree Double
t3 = Node "abc" (Node "b" Nil Nil) (Node "xyz" Nil Nil)
                                 :: Tree [Char]
```

```
newtype Ptype a b c = P (a,(b,b),[c],(a->b->c))
pt = P (2,("hello","world"),[True,False,True],
        \x->(\y->(x > length y)))
                              :: Ptype Int [Char] Bool
```

Heterogene algebraische Typen

Beispiel: Heterogene Bäume

```
data HTree = LeafS String
           | LeafF (Int -> Int)
           | NodeF (String -> Int -> Bool) HTree HTree
           | NodeM Bool Float Char HTree HTree
```

Zwei Varianten der Funktion Tiefe auf Werten vom Typ HTree:

```
depth :: HTree -> Int
depth (LeafS _)           = 1
depth (LeafF _)           = 1
depth (NodeF _ t1 t2)     = 1 + max (depth t1) (depth t2)
depth (NodeM _ _ _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: HTree -> Int
depth (NodeF _ t1 t2)     = 1 + max (depth t1) (depth t2)
depth (NodeM _ _ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _                   = 1
```

Polymorphe heterogene algebraische Typen

...sind genauso möglich, z.B. heterogene polymorphe Bäume:

```
data PHTree a b c d
        = LeafA a b b c c c (a->b)
          | LeafB [b] [(a->b->c->d)]
          | NodeC (c,d) (PHTree a b c d) (PHTree a b c d)
          | NodeD [(c,d)] (PHTree a b c d) (PHTree a b c d)
```

Zwei Varianten der Fkt. Tiefe auf Werten vom Typ PHTree:

```
depth :: (PHTree a b c d) -> Int
depth (LeafA _ _ _ _ _ _ _) = 1
depth (LeafB _ _)           = 1
depth (NodeC _ t1 t2)       = 1 + max (depth t1) (depth t2)
depth (NodeD _ t1 t2)       = 1 + max (depth t1) (depth t2)

depth :: (PHTree a b c d) -> Int
depth (NodeC _ t1 t2)       = 1 + max (depth t1) (depth t2)
depth (NodeD _ t1 t2)       = 1 + max (depth t1) (depth t2)
depth _                     = 1
```

Polymorphe Typsynonyme

Typsynonyme und Funktionen darauf dürfen polymorph sein:

Beispiel:

```
type Sequence a = [a]

lengthSeq :: Sequence a -> Int
lengthSeq []     = 0
lengthSeq (_:xs) = 1 + lengthList xs
```

bzw. knapper:

```
lengthSeq :: Sequence a -> Int
lengthSeq = length
```

Beachte: Abstützen auf Standardfunktion `length` ist möglich, da `Sequence a` Typsynonym ist, kein neuer Typ.

Polymorphe newtype-Deklarationen

newtype-Deklarationen und Funktionen auf solchen Typen dürfen polymorph sein:

Beispiel:

```
newtype Ptype a b c = P (a,(b,b),[c],(a->b->c))

pt = P (2,("hello","world"),[True,False,True],
        \x->(\y->(x > length y)))
                          :: Ptype Int [Char] Bool

f :: String -> (Ptype a b c) -> (Ptype a b c)
f s pt = ...
```

Typklassen in Haskell vs. Klassen in objektorientierten Sprachen

Klassen in objektorientierten Programmiersprachen

- dienen der Strukturierung von Programmen.
- liefern Blaupausen zur Generierung von Objekten mit bestimmten Fähigkeiten.

(Typ-) Klassen in Haskell

- dienen nicht der Strukturierung von Programmen, sondern fassen Typen mit ähnlichem Verhalten zusammen, d.h. deren Werte sich vergleichbar manipulieren lassen.
- liefern keine Blaupausen zur Generierung von Werten, sondern werden vorab definiert und anschließend entsprechend ihres Verhaltens passenden bereits existierenden oder neuen Typklassen durch automatisch (`deriving`) oder explizite (`instance`) Instanzbildung zugeordnet.

Zusammenfassung

Wir halten fest:

- Datenstrukturen können "mehrfach" polymorph sein.
- Polymorphe Heterogenität ist für
  - `data`-,
  - `newtype`- und
  - `type`-Deklarationen

  in gleicher Weise möglich.

Kapitel 8.3

Zusammenfassung und Resümee

Polymorphie auf Typen und Funktionen (1)

...unterstützt

- Wiederverwendung durch Parametrisierung!

Ermöglicht durch:

- Die bestimmenden Eigenschaften eines Datentyps sind wie die bestimmenden Eigenschaften darauf arbeitender Funktionen oft unabhängig von bestimmten typspezifischen Details.

Insgesamt: Ein typisches Vorgehen in der Informatik

- durch Parametrisierung werden gleiche Teile "ausgeklammert" und damit der Wiederverwendung zugänglich!
- (i.w.) gleiche Codeteile müssen nicht (länger) mehrfach geschrieben werden.

Polymorphie auf Typen und Funktionen (2)

Polymorphie und die von ihr ermöglichte Wiederverwendung unterstützt somit die

- Ökonomie der Programmierung (flapsig: "*Schreibfaulheit*")

Insbesondere trägt Polymorphie bei zu höherer

- Transparenz und Lesbarkeit
  ...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit
  ...ein Aspekt mit mehreren Dimensionen wie Fehlersuche, Weiterentwicklung, etc.; hier ein willkommener Nebeneffekt!
- ...
- Effizienz (der Programmierung)
  ...höhere Produktivität, früherer Markteintritt (time-to-market)

Polymorphie auf Typen und Funktionen (3)

Auch in anderen Paradigmen

- ...wie etwa imperativer und objektorientierter Programmierung lernt man, den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!

  Aktuelles Stichwort: Generic Java

Fazit über Teil III "Fkt. Programmierung" (1)

Die Stärken des funktionalen Programmierstils

- resultieren insgesamt aus wenigen Konzepten
  - sowohl bei Funktionen
  - als auch bei Datentypen

Schüsselrollen spielen die Konzepte von

- Funktionen als first class citizens
  - Funktionen höherer Ordnung
- Polymorphie auf
  - Funktionen
  - Datentypen

Fazit über Teil III “Fkt. Programmierung” (2)

Die Ausdruckskraft und Flexibilität des funktionalen Programmierstils

- ergibt sich insgesamt durch die Kombination und das nahtlose Zusammenspiel der tragenden wenigen Einzelkonzepte.

  $\rightsquigarrow$ *das Ganze ist mehr als die Summe seiner Teile!*

Fazit über Teil III “Fkt. Programmierung” (3)

Speziell in Haskell tragen zur Ausdruckskraft und Flexibilität darüberhinaus auch sprachspezifische Annehmlichkeiten bei, insbesondere zur automatischen Generierung, etwa von

- Listen: `[2,4..42]`, `[odd n | n <- [1..], n<1000]`
- Selektorfunktionen: Verbundtyp-Syntax für algebraische Datentypen
- Instanzbildungen: `deriving`-Klausel

Für eine vertiefende und weiterführende Diskussion siehe:

- Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003.

Kapitel 8.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 8 (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, 2007. (Kapitel 3, Types and classes; Kapitel 10, Declaring types and classes)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 8 (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 8 (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 IV

Fundierung funktionaler Programmierung

Kapitel 9

Auswertungsstrategien

Auswertungsstrategien

...für Ausdrücke:

- Applikative Auswertungsordnung (applicative order)
  - *Verwandte Ausdrücke:* call-by-value Auswertung, leftmost-innermost Auswertung, strikte Auswertung, eager evaluation
- Normale Auswertungsordnung (normal order)
  - *Verwandte Ausdrücke:* call-by-name Auswertung, leftmost-outermost Auswertung
  - *Verwandte Strategie:* lazy evaluation
    - *Verwandter Ausdruck:* call-by-need Auswertung

Auswertungsstrategien (fgs.)

Zentral für alle Strategien:

Die Organisation des Zusammenspiels von

- Expandieren ($\rightsquigarrow$ Funktionsaufrufe)
- Simplifizieren ($\rightsquigarrow$ einfache Ausdrücke)

um einen Ausdruck soweit zu vereinfachen wie möglich.

Kapitel 9.1

Einführende Beispiele

Auswerten von Ausdrücken

Drei Beispiele:

1. Arithmetischer Ausdruck:

```
3 * (9+5) ->> ...
```

2. Ausdruck mit Aufruf nichtrekursiver Funktion:

```
simple :: Int -> Int -> Int -> Int
simple x y z = (x+z) * (y+z)

simple 2 3 4 ->> ...
```

3. Ausdruck mit Aufruf rekursiver Funktion:

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

fac 2 ->> ...
```

Beispiel 1): 3 * (9+5)

Viele Simplifikations-Wege führen zum Ziel:

Simplifikations-Weg 1:
```
3 * (9+5)
  (Simplifizieren) ->> 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
```

Beispiel 2a): `simple 2 3 4`

```
simple x y z ::  Int -> Int -> Int
simple x y z = (x + z) * (y + z)
```

ES-Weg 1:

```
simple 2 3 4
  (Expandieren) ->> (2 + 4) * (3 + 4)
  (Simplifizieren) ->> 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 ->> ...
```

Beispiel 2b): `simple 2 3 ((5+7)*9)`

```
simple x y z ::  Int -> Int -> Int
simple x y z = (x + z) * (y + z)
```

ES-Weg 1: `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: `simple 2 3 ((5+7)*9)`

```
    (E) ->> (2 + ((5+7)*9)) * ((3 + (5+7)*9))
    (S) ->> ...
    (S) ->> 12.210
```

- Weg 1: Applikative Auswertung
- Weg 2: Normale Auswertung

Beispiel 3): `fac 2`

```
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)
```

Für die Fortführung der Berechnung

- gibt es auch hier die Möglichkeiten
  - applikativ
  - normal

  fortzufahren.

Wir nutzen diese Freiheitsgrade aus

- und verfolgen beide Möglichkeiten im Detail

Beispiel 3): `fac 2`

Applikativ:

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

in diesem Stil fortfahren

Normal:

```
          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))
        ->> ...
```

in diesem Stil fortfahren

Beispiel 3): Applikative Auswertung

```
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
```

⇝ sog. eager (sofortige) evaluation

Beispiel 3): Normale Auswertung

```
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)
   (E) ->> 2 * (if (2-1) == 0 then 1
                     else ((2-1) * fac ((2-1)-1)))
  (3S) ->> 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)))
  (4S) ->> 2 * (1 * 1)
   (S) ->> 2 * 1
   (S) ->> 2
```

⇝ Intelligent umgesetzt: sog. lazy (verzögerte) evaluation

Weitere Freiheitsgrade

Betrachte:

```
2*3+fac(fib(square(2+2)))+3*5+fib(fac((3+5)*7))+5*7
```

Zwei Freiheitsgrade:

- Wo im Ausdruck mit der Auswertung fortfahren?
- Wie mit (Funktions-) Argumenten umgehen?

Zentrale Frage:

- Was ist der Einfluss auf das Ergebnis?

Hauptresultat (im Vorgriff)

Theorem

Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

Alonzo Church, John Barkley Rosser (1936)

Beachte: Angesetzt auf denselben Ausdruck mögen einige Auswertungsreihenfolgen terminieren, andere nicht (Beispiele später in diesem Kapitel). Diejenigen, die terminieren, terminieren mit demselben Ergebnis.

Kapitel 9.2

Applikative und normale Auswertungsordnung

Applikative und normale Auswertungsordnung

...sind zwei für die Praxis besonders wichtige Auswertungsstrategien:

- Applikative Auswertungsordnung
  (engl. applicative order evaluation)
  - Umsetzung: Eager evaluation (sofortige Auswertung)
- Normale Auswertungsordnung
  (engl. normal order evaluation)
  - Intelligente Umsetzung: Lazy evaluation (verzögerte Auswertung)

Applikative und normale Auswertungsordnung unterscheiden sich besonders

- in der Auswertungsphilosophie für Funktionsaufrufe: `f e`

Applikative Auswertungsordnung

Applikativ:

- Um den Ausdruck `f e` auszuwerten:
  - berechne zunächst den Wert `w` von `e` und setze diesen Wert `w` dann im Rumpf von `f` ein

    (applicative order evaluation, call-by-value evaluation, leftmost-innermost evaluation, strict evaluation, eager evaluation)

Normale Auswertungsordnung

Normal:

- Um den Ausdruck `f e` auszuwerten:
  - setze `e` unmittelbar (d.h. unausgewertet) im Rumpf von `f` ein und werte den so entstehenden Ausdruck aus

    (normal order evaluation, call-by-name evaluation, leftmost-outermost evaluation.

    Intelligente Umsetzung: lazy evaluation, call-by-need evaluation)

Beispiele: Einige einfache Funktionen

Die Funktion square zur Quadrierung einer ganzen Zahl

```
square :: Int -> Int
square n = n * n
```

Die Funktion first zur Projektion auf die erste Paarkomponente

```
first :: (Int,Int) -> Int
first (m,n) = m
```

Die Funktion infiniteInc zum "ewigen" Inkrement

```
infiniteInc :: Int
infiniteInc = 1 + infiniteInc
```

Ausw. in applikativer Auswertungsordnung

...leftmost-innermost (LI) evaluation:

```
              square (square (square (1+1)))
  (LI-S) ->> square (square (square 2))
  (LI-E) ->> square (square (2*2))
  (LI-S) ->> square (square 4)
  (LI-E) ->> square (4*4)
  (LI-S) ->> square 16
  (LI-E) ->> 16*16
  (LI-S) ->> 256
```

Insgesamt: 7 Schritte.

Bemerkung: (LI-E): LI-Expansion / (LI-S): LI-Simplifikation

Ausw. in normaler Auswertungsordnung

...leftmost-outermost (LO) evaluation:

```
           square (square (square (1+1)))
(LO-E) ->> square (square (1+1)) * square (square (1+1))
(LO-E) ->> ((square (1+1))*(square (1+1))) * square (square (1+1))
(LO-E) ->> (((1+1)*(1+1))*square (1+1)) * square (square (1+1))
(LO-S) ->> ((2*(1+1))*square (1+1)) * square (square (1+1))
(LO-S) ->> ((2*2)*square (1+1)) * square (square (1+1))
(LO-S) ->> (4 * square (1+1)) * square (square (1+1))
(LO-E) ->> (4*((1+1)*(1+1))) * square (square (1+1))
(LO-S) ->> (4*(2*(1+1))) * square (square (1+1))
(LO-S) ->> (4*(2*2)) * square (square (1+1))
(LO-S) ->> (4*4) * square (square (1+1))
(LO-S) ->> 16 * square (square (1+1))
       ->> ...
(LO-S) ->> 16 * 16
(LO-S) ->> 256
```

Insgesamt: 1+10+10+1=22 Schritte.

Bemerkung: (LO-E): LO-Expansion / (LO-S): LO-Simplifikation

Applikative Auswertungsordnung effizienter?

Nicht immer; betrachte:

```
first (2*21, square (square (square (1+1))))
```

- In applikativer Auswertungsordnung:

```
    first (2*21, square (square (square (1+1))))
->> first (42, square (square (square (1+1))))
->> ...
->> first (42, 256)
->> 42
```

  Insgesamt: $1+7+1=9$ Schritte.

- In normaler Auswertungsordnung:

```
    first (2*21, square (square (square (1+1))))
->> 2*21
->> 42
```

  Insgesamt: 2 Schritte. (Das zweite Argument wird nicht benötigt und auch nicht ausgewertet!)

Applikative vs. normale Auswertung (1)

Das Hauptresultat von Church und Rosser garantiert:

- Terminieren applikative und normale Auswertungsordnung angewendet auf einen Ausdruck beide, so terminieren sie mit demselben Resultat.

Aber:

Applikative und normale Auswertungsordnung können sich unterscheiden

- in der Zahl der Schritte bis zur Terminierung (mit gleichem Resultat)
- im Terminierungsverhalten
  - Applikativ: Nichttermination, kein Resultat: undefiniert
  - Normal: Termination, sehr wohl ein Resultat: definiert

(Bem.: Die umgekehrte Situation ist nicht möglich!)

Betrachte hierzu folgendes Beispiel:

```
first (2*21, infiniteInc)
```

Applikative vs. normale Auswertung (2)

In applikativer Auswertungsordnung:

```
    first (2*21, infiniteInc)
->> first (42, infiniteInc)
->> first (42, 1+infiniteInc)
->> first (42, 1+(1+infiniteInc))
->> first (42, 1+(1+(1+infiniteInc)))
->> ...
->> first (42, 1+(1+(1+(...+(1+infiniteInc)...))))
->> ...
```

Insgesamt: Nichtterminierung, kein Resultat: undefiniert!

Applikative vs. normale Auswertung (3)

In normaler Auswertungsordnung:

```
    first (2*21, infiniteInc)
->> 2*21
->> 42
```

Insgesamt: Terminierung, Resultat nach 2 Schritten: definiert!

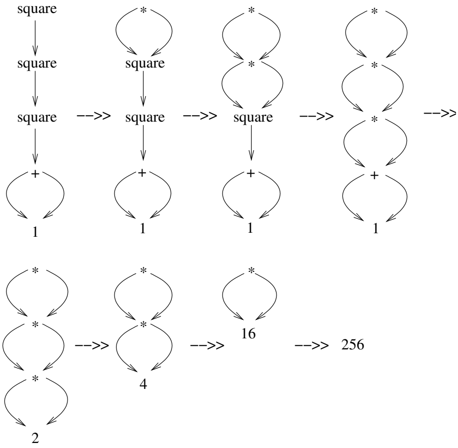
Normale Auswertungsordnung intelligent

- Problem: Bei normaler Auswertungsordnung erfolgt häufig Mehrfachauswertung von Ausdrücken (siehe etwa Beispiel 2b), Weg 2), oder normale Auswertung von `square (square (square (1+1)))`
- Ziel: Vermeidung von Mehrfachauswertungen zur Effizienzsteigerung
- Methode: Darstellung von Ausdrücken in Form von Graphen, in denen gemeinsame Teilausdrücke geteilt sind; Auswertung von Ausdrücken direkt in Form von Transformationen dieser Graphen.

- Resultierende Auswertungsstrategie: Lazy Evaluation!

  ...garantiert, dass Argumente höchstens einmal ausgewertet werden (möglicherweise also gar nicht!).

Termrepräsentation und -transformation auf Graphen



Insgesamt: 7 Schritte.

(runter von 22 Schritten für (naive) normale Auswertung)

Lazy Evaluation

In Summe:

Lazy evaluation bzw. verzögerte Auswertung

- ist eine intelligente und effiziente Umsetzung der normalen Auswertungsordnung.
- beruht implementierungstechnisch auf Graphdarstellungen von Ausdrücken und Graphtransformationen zu ihrer Auswertung statt auf Termdarstellungen und Termtransformationen.
- "vergleichbar" performant wie applikative (eager) Auswertungsordnung, falls alle Argumente benötigt werden.
- vereint möglichst gut die Vorteile von applikativer (Effizienz!) und normaler (Terminierungshäufigkeit!) Auswertungsordnung.

Hauptresultate

Theorem

1. *Alle terminierenden Auswertungsreihenfolgen enden mit demselben Ergebnis*
   $\rightsquigarrow$ *Konfluenz- oder Diamanteigenschaft*
2. *Wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normale Auswertungsreihenfolge*
   $\rightsquigarrow$ *Standardisierungstheorem*

Alonzo Church, John Barkley Rosser (1936)

Wichtig:

- Teilaussage 2) des obigen Theorems gilt in gleicher Weise für die lazy Auswertungsordnung

Informell bedeutet das:

- Lazy evaluation (und normale Auswertungsordnung) terminieren am häufigsten, so oft wie überhaupt möglich.

Kapitel 9.3

Eager oder Lazy Evaluation? Eine Abwägung

Frei nach Shakespeare

"Eager or lazy evaluation?

...that is the question".

Quot capita, tot sensa —
die Meinungen sind verschieden:

- Eager evaluation
  (z.B. in ML, Scheme (abgesehen von Makros),...)
- Lazy evaluation
  (z.B. in Haskell, Miranda,...)

Eager vs. Lazy Evaluation: Eine Abwägung (1)

Lazy Evaluation

- Stärken
  - Terminiert mit Normalform, wenn es (irgend-) eine terminierende Auswertungsreihenfolge gibt.

    Informell: Lazy (und normale) Auswertungsordnung terminieren am häufigsten, so oft wie überhaupt möglich!
  - Wertet Argumente nur aus, wenn nötig; und dann nur einmal.
  - Ermöglicht eleganten und flexiblen Umgang mit möglicherweise unendlichen Werten von Datenstrukturen (z.B. unendliche Listen, unendliche Bäume, etc.).

Eager vs. Lazy Evaluation: Eine Abwägung (2)

Lazy Evaluation

- Schwächen
  - Konzeptuell und implementierungstechnisch anspruchsvoller
    - Graph- statt Termrepräsentationen und -transformationen
    - Partielle Auswertung von Ausdrücken: Seiteneffekte!

      (Beachte: Seiteneffekte nicht in Haskell! In Scheme: Verantwortung liegt beim Programmierer.)
    - Ein-/Ausgabe nicht in trivialer Weise transparent für den Programmierer zu integrieren
    - Volle Einsicht erfordert profundes Verständnis von Bereichstheorie (domain theory) und λ-Kalkül

Eager vs. Lazy Evaluation: Eine Abwägung (3)

Eager Evaluation

- Stärken:
  - Konzeptuell und implementierungstechnisch einfacher
  - Vom mathematischen Standpunkt oft "natürlicher" (Beispiel: `first (2*21,infiniteInc)`)
  - Einfache(re) Integration imperativer Konzepte

Resümee

Eager or lazy evaluation

- Für beide Strategien sprechen gute Gründe

Somit:

- Die Wahl ist eine Frage des Anwendungskontexts!

Zur Ersetzbarkeit von lazy durch eager (1)

Wäre ein Haskell-Compiler (Interpretierer) korrekt, der die Fakultätsfunktion applikativ auswertete?

- Ja, weil die Funktion `fac` strikt in ihrem Argument ist.

Denn:

Für strikte Funktionen stimmen

- das Terminierungsverhalten von eager und lazy Auswertungsordnung überein.

Nach dem Konfluenztheorem von Church und Rosser stimmen zusätzlich dann auch

- die Resultate von von eager und lazy Auswertungsordnung überein.

Zur Ersetzbarkeit von lazy durch eager (2)

Deshalb darf für strikte Funktionen wie z.B. `fac`

- lazy durch eager Auswertung ersetzt werden, da weder das Resultat noch das Terminierungsverhalten geändert wird.

Statt von eager evaluation spricht man deshalb manchmal auch von

- strikter Auswertung (strict evaluation)!

Strikte Funktionen (1)

Eine Funktion heißt strikt

- in einem Argument, wenn: Ist das Argument nicht definiert, so ist auch der Wert der Funktion für dieses Argument nicht definiert.
  - Beispiel: Die Fakultätsfunktion oder die Fibonacci-Funktion.

Mehrstellige Funktionen können

- strikt sein in einigen Argumenten, nicht strikt in anderen.
  - Beispiel: Der Fallunterscheidungsausdruck
    `(if . then . else .)`
    ist strikt im ersten Argument (Bedingung), nicht aber im zweiten (then-Ausdruck) und dritten (else-Ausdruck).

Strikte Funktionen (2)

Die Ersetzung von verzögerter (lazy) durch strikte Auswertung, wo möglich, d.h. ohne Änderung des Terminierungsverhaltens, ist

- wichtige Optimierung bei der Übersetzung funktionaler Programme.

Auswertungsordnungen im Vergleich (1)

...über Analogien und Betrachtungen zu

(i) Parameterübergabemechanismen
(ii) Auswertungsposition
(iii) Häufigkeit von Argumentauswertungen

(i) Auswertungsordnungen im Vergleich (2)

...über Analogien zu Parameterübergabemechanismen:

- Normale Auswertungsordnung
  - Call-by-name
- Applikative Auswertungsordnung
  - Call-by-value
- Lazy Auswertungsordnung
  - Call-by-need

(ii) Auswertungsordnungen im Vergleich (3)

...über die Position der Auswertung im Ausdruck:

- Outermost-Auswertungsordnung: Reduziere nur Redexe, die nicht in anderen Redexen enthalten sind

  Leftmost-Auswertungsordnung: Reduziere stets den linkesten Redex

  - entsprechen normaler Auswertungsordnung
- Innermost-Auswertungsordnung: Reduziere nur Redexe, die keine Redexe enthalten
  - entspricht applikativer Auswertungsordnung

(ii) Auswertungsordnungen im Vergleich (3)

...über die Position der Auswertung im Ausdruck:

- Leftmost-outermost Auswertungsordnung
  - Spezielle normale Auswertungsordnung: in optimierter Implementierungsform sog. lazy Auswertung
- Leftmost-innermost-Auswertungsordnung
  - spezielle applikative Auswertungsordnung: sog. eager Auswertung

(iii) Auswertungsordnungen im Vergleich (4)

...über die Häufigkeit von Argumentauswertungen:

- Normale Auswertungsordnung
  - Argumente werden so oft ausgewertet, wie sie benutzt werden
- Applikative Auswertungsordnung
  - Argumente werden genau einmal ausgewertet
- Lazy Auswertungsordnung
  - Argumente werden höchstens einmal ausgewertet

Veranschaulichung

Betrachte die Funkion

```
f :: Integer -> Integer -> Integer -> Integer
f x y z = if x>42 then y*y else z+z
```

und den Aufruf

```
f 45 (square (5*(2+3))) (square ((2+3)*7))
```

a) Applikativ ausgewertet

```
f x y z = if x>42 then y*y else z+z
```

Applikative Auswertung

```
f 45 (square (5*(2+3))) (square ((2+3)*7)))
(S) ->> f 45 (square (5*5)) (square (5*7))
(S) ->> f 45 (square 25) (square 35)
(S) ->> ...
(S) ->> f 45 625 1.225
(E) ->> if 45>42 then 625*625 else 1.125*1.125
(S) ->> if True then 625*625 else 1.125*1.125
(S) ->> 625*625
(S) ->> 390.625
```

...die Argumente `(square (5*(2+3)))` und `(square ((2+3)*7)))` werden beide genau einmal ausgewertet.

b) Normal ausgewertet

```
f x y z = if x>42 then y*y else z+z
```

Normale Auswertung

```
f 45 (square (5*(2+3))) (square ((2+3)*7)))
(E) ->> if 45>42 then (square (5*(2+3))) * (square (5*(2+3)))
        else (square ((2+3)*7))) + (square ((2+3)*7)))
(S) ->> if True then (square (5*(2+3))) * (square (5*(2+3)))
        else (square ((2+3)*7))) + (square ((2+3)*7)))
(S) ->> (square (5*(2+3))) * (square (5*(2+3)))
(S) ->> (square (5*5)) * (square (5*5))
(S) ->> (square 25) * (square 25)
(S) ->> 625 * 625
(S) ->> 390.625
```

...das Argument `(square (5*(2+3)))` wird zweimal ausgewertet; das Argument `(square ((2+3)*7)))` gar nicht.

c) Lazy ausgewertet

```
f x y z = if x>42 then y*y else z+z
```

Lazy Auswertung

```
f 45 (square (5*(2+3))) (square ((2+3)*7)))
(E) ->> if > then   *     else   +
          / \      / \          / \
         45 42     \ /          \ /
                 square       square
                    |            |
                    *            *
                   / \ ________/ \
                  5   +            7
                     / \
                    2   3
   ->> ... ->> 390.625
```

...das Argument (square (5*(2+3))) wird genau einmal ausgewertet; das Argument (square ((2+3)*7))) gar nicht.

Zusammenfassende Abwägung (1)

- Normale Auswertungsordnung
  - Argumente werden so oft ausgewertet, wie sie benutzt werden
    - \+ Kein Argument wird ausgewertet, dessen Wert nicht benötigt wird
    - \+ Terminiert häufiger als applikative Auswertung
    - – Argumente, die mehrfach benötigt werden, werden auch mehrfach ausgewertet
- Applikative Auswertungsordnung
  - Argumente werden genau einmal ausgewertet
    - \+ Argumente werden exakt einmal ausgewertet; kein zusätzlicher Aufwand über die Auswertung hinaus
    - – Argumente werden auch dann ausgewertet, wenn sie nicht benötigt werden; dies ist kritisch für Argumente, deren Auswertung teuer ist, auf einen Laufzeitfehler führt oder nicht terminiert

Zusammenfassende Abwägung (2)

- Lazy Auswertungsordnung
  - Argumente werden höchstens einmal ausgewertet
    - \+ Ein Argument wird nur ausgewertet, wenn sein Wert benötigt wird; und dann nur genau einmal
    - \+ Kombiniert die Vorteile von applikativer (Effizienz) und normaler (Terminierung) Auswertung
    - – Erfordert zur Laufzeit zusätzlichen Aufwand zur Verwaltung der Auswertung von (Teil-) Ausdrücken

Zusammenfassende Abwägung (3)

...von einem pragmatischen Standpunkt aus:

- Applikative Auswertungsordnung vorteilhaft gegenüber normaler und lazy Auswertungsordnung, da
  - geringere Laufzeitzusatzkosten (Overhead)
  - größeres Parallelisierungspotential (für Funktionsargumente)
- Lazy Auswertungsordnung vorteilhaft gegenüber applikativer Auswertungsordnung, wenn
  - Terminierungshäufigkeit (Definiertheit des Programms!) zentral
  - Argumente nicht benötigt (und deshalb gar nicht ausgewertet) werden (Beispiel: $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))z$)
- "Ideale" Auswertungsordnung
  - Das Beste beider Welten:
    Applikativ, wo möglich; lazy, wo nötig
    (Beispiel: Fakultätsfunktion).

Kapitel 9.4

Eager und Lazy Evaluation in Haskell

Auswertungssteuerung in Haskell (1)

Haskell erlaubt, die Auswertungsordnung (zu einem gewissen Grad) zu steuern.

Lazy Auswertung:

- Standardverfahren (vom Programmierer nichts zu tun).

Eager-ähnliche Auswertung:

- Mithilfe des zweistelligen Operators `$!`
  Beispiel:

```
        fac (2*(3+5))
(E) ->> if (2*(3+5)) == 0 then 1
        else ((2*(3+5)) * fac ((2*(3+5))-1))
...
        fac $! (2*(3+5))
(S) ->> fac (2*8)
(S) ->> fac 16
(E) ->> if 16 == 0 then 1 else (16 * fac (16-1))
...
```

Auswertungssteuerung in Haskell (2)

Detaillierter:

- Die Auswertung eines Ausdrucks `f $! x` erfolgt in gleicher Weise wie die Auswertung des Ausdrucks `f x` mit dem Unterschied, dass die Auswertung von `x` erzwungen wird, bevor `f` angewendet wird.

Wirkung: Ist das Argument `x` von einem

- elementaren Typ wie `Int`, `Bool`, `Double`, etc., so wird `x` vollständig ausgewertet.
- Tupeltyp wie `(Int,Bool)`, `(Int,Bool,Double)`, etc., so wird `x` bis zu einem Tupel von Ausdrücken ausgewertet, aber nicht weiter.
- Listentyp, so wird `x` so weit ausgewertet, bis als Ausdruck die leere Liste erscheint oder die Konstruktion zweier Ausdrücke zu einer Liste.

Auswertungssteuerung in Haskell (3)

- In Anwendung mit einer curryfizierten Funkion `f` kann mittels `$!` strikte Auswertung für jede Argumentkombination erreicht werden:

  Beispiel:

  Für zweistelliges `f :: a -> b -> c`
  - `(f $! x) y`: erzwingt Auswertung von `x`
  - `(f x) $! y`: erzwingt Auswertung von `y`
  - `(f $! x) $! y`: erzwingt Auswertung von `x` und `y`

  vor Anwendung von `f`

Anwendungsbeispiel (1)

Hauptanwendung von `$!` in Haskell:

- Zur Speicherverbrauchsverminderung

Beispiel:

```
lz_sumwith :: Int -> [Int] -> Int
lz_sumwith v []     = v
lz_sumwith v (x:xs) = lz_sumwith (v+x) xs
```

versus

```
ea_sumwith :: Int -> [Int] -> Int
ea_sumwith v []     = v
ea_sumwith v (x:xs) = (ea_sumwith $! (v+x)) xs
```

Anwendungsbeispiel (2)

Lazy Auswertung ergibt:

```
        lz_sumwith 5 [1,2,3]
(E) ->> lz_sumwith (5+1) [2,3,]
(E) ->> lz_sumwith ((5+1)+2) [3]
(E) ->> lz_sumwith (((5+1)+2)+3) []
(E) ->> (((5+1)+2)+3)
(S) ->> ((6+2)+3)
(S) ->> (8+3)
(S) ->> 11
```

$\rightsquigarrow$ 7 Schritte

Anwendungsbeispiel (3)

Eager Auswertung ergibt:

```
         ea_sumwith 5 [1,2,3]
(E) ->> (ea_sumwith $! (5+1)) [2,3]
(S) ->> (ea_sumwith $! 6) [2,3]
(S) ->> ea_sumwith 6 [2,3]
(E) ->> (ea_sumwith $! (6+2)) [3]
(S) ->> (ea_sumwith $! 8) [3]
(S) ->> ea_sumwith 8 [3]
(E) ->> (ea_sumwith $! (8+3)) []
(S) ->> (ea_sumwith $! 11) []
(S) ->> ea_sumwith 11 []
(E) ->> 11
```

$\leadsto$ 10 Schritte

Anwendungsbeispiel (4)

Beobachtung:

- Lazy Auswertung von `lz_sumwith 5 [1,2,3]`
  - 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 Zahl der Elemente in der Argumentliste ist
    - Problem: Programmabbrüche durch Speicherüberläufe können schon bei vergleichsweise kleinen Argumenten auftreten: `lz_sumwith 5 [1..10000]`
- Eager Auswertung von `ea_sumwith 5 [1,2,3]`
  - Simplifikationen werden frühestmöglich ausgeführt
  - Exzessiver Speicherverbrauch (engl. memory leaks) tritt nicht auf
    - Aber: Die Zahl der Rechenschritte steigt: Besseres Speicherverhalten wird gegen schlechtere Schrittzahl eingetauscht (trade-off)

Schlussbemerkung

Naive Anwendung des `$!`-Operators in Haskell ist

- kein Königsweg, das Speicherverhalten zu verbessern
- erfordert (bereits bei kleinen Beispielen) sorgfältige Untersuchung des Verhaltens der lazy Auswertung

Übersetzer führen 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,

- lazy

durch

- eager

Auswertung zu ersetzen.

Kapitel 9.5

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (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, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.2, Models of Reduction; Kapitel 6.3, Reduction Order and Space)
- Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 2.3, Reduction Strategies)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (2)

- 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, 2007. (Kapitel 12, Lazy Evaluation; Kapitel 12.2, Evaluation Strategies; Kapitel 12.7, Strict Application)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 9 (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 9 (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 9 (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 10

λ-Kalkül

"...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)

Kapitel 10.1

Hintergrund und Motivation: Berechenkarkeitstheorie und Berechenbarkeitsmodelle

λ-Kalkül

Der λ-Kalkül

- ist zusammen mit
  - Turing-Maschinen
  - Markov-Algorithmen
  - Theorie rekursiver Funktionen

  (und weiteren formalen Berechenbarkeitsmodellen) fundamental für die Berechenbarkeitstheorie.
- liefert formale Fundierung funktionaler Programmiersprachen

Berechenbarkeitstheorie

Im Mittelpunkt stehende Fragen:

- Was heißt berechenbar?
- Was ist berechenbar?
- Wie aufwändig ist etwas zu berechnen?
- Gibt es Grenzen der Berechenbarkeit?
- ...

Informeller Berechenbarkeitsbegriff

Ausgangspunkt:

- eine informelle Vorstellung von Berechenbarkeit

Daraus resultierend:

- 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 die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht.

Intuitive Berechenbarkeit

Frage:

- Was ist mit dieser informellen Annäherung an den Begriff der Berechenbarkeit gewonnen?

Antwort:

- Für die Beantwortung der konkreten Fragen der Berechenbarkeitstheorie zunächst einmal nichts, da der Begriff intuitiv berechenbar vollkommen vage und nicht greifbar ist:

  "...eine irgendwie machbare effektive mechanische Methode..."

Formale Berechenbarkeit

Zentrale Aufgabe der Berechenbarkeitstheorie:

- den Begriff der Berechenbarkeit formal zu fassen und ihn so einer präzisen Behandlung zugänglich zu machen.

Das erfordert:

- Formale Berechnungsmodelle, d.h. Explikationen des Begriffs "intuitiv berechenbar"

Der λ-Kalkül

...ist ein solches formales Berechnungsmodell.

Ebenso wie

- Turing-Maschinen
- Markov-Algorithmen
- Theorie rekursiver Funktionen

...und eine Reihe weiterer Ausprägungen formaler Berechenbarkeitsmodelle.

Vergleich von Berechenbarkeitsmodellen

Das Berechenbarkeitsmodell der

- Turing-Maschinen

ist eine maschinen-basierte und -orientierte Präzisierung des Berechenbarkeitsbegriffs.

Die Berechenbarkeitsmodelle der

- Markov-Algorithmen
- Theorie rekursiver Funktionen
- λ-Kalkül

sind eine programmier-basierte und -orientierte Präzisierung des Berechenbarkeitsbegriffs.

Der λ-Kalkül

...ist über die Präzisierung des Berechenbarkeitsbegriffs hinaus besonders wichtig und nützlich für:

- Design von Programmiersprachen und Programmiersprachkonzepten
  - Speziell funktionale Programmiersprachen
  - Speziell Typsysteme und Polymorphie
- Semantik von Programmiersprachen
  - Speziell denotationelle Semantik und Bereichstheorie (engl. domain theory)
- Berechenbarkeitstheorie
  - Speziell Grenzen der Berechenbarkeit

Der λ-Kalkül im Überblick

Der λ-Kalkül

- geht zurück auf Alonzo Church (1936)
- ist spezielles formales Berechnungsmodell, wie viele andere auch, z.B.
  - 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)
  - ...
- formalisiert Berechnungen über Paaren, Listen, Bäumen, auch möglicherweise unendlichen, über Funktionen höherer Ordnung, etc., und macht sie einfach ausdrückbar
- ist in diesem Sinne "praxisnäher/realistischer" als (manche) andere formale Berechnungsmodelle

Die Church'sche These (1)

Church'sche These

Eine Funktion ist genau dann intuitiv berechenbar, wenn sie λ-definierbar ist (d.h. im λ-Kalkül ausdrückbar ist).

Beweis? Umöglich!

Die Church'sche These (2)

Die Church'sche These entzieht sich wg. der grundsätzlichen Nichtfassbarkeit des Begriffs intuitiv berechenbar jedem Beweisversuch.

Man hat jedoch folgendes bewiesen:

- Alle der obigen (und die weiters vorgeschlagenen) formalen Berechnungsmodelle sind gleich mächtig.

Dies kann als starker Hinweis darauf verstanden werden, dass

- alle diese formalen Berechnungsmodelle den Begriff wahrscheinlich "gut" charakterisieren!

Die Church'sche These (3)

Aber: Dieser starke Hinweis schließt nicht aus, dass morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit "besser" charakterisierte.

Präzedenzfall: Primitiv rekursive Funktionen

- bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit akzeptiert
- tatsächlich jedoch: echt schwächeres Berechnungsmodell
- Beweis: Ackermann-Funktion ist berechenbar, aber nicht primitiv rekursiv (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 zweifellos

- berechenbaren, deshalb insbesondere intuitiv berechenbaren, jedoch nicht primitiv rekursiven Funktion!

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 der

- angemessenen Formalisierung des Begriffs intuitiver Berechenbarkeit

...ist die Frage nach der

- Angemessenheit des Begriffs 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 die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht."

induziert eine

- funktionsorientierte Vorstellung von Algorithmus

die Berechenbarkeitsformalisierungen wie dem λ-Kalkül und anderen zugrundeliegt und weitergehend implizit die Problemtypen festlegt, die überhaupt als

- Berechenbarkeitsproblem

aufgefasst werden.

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)

Interaktion zwischen Anwender und Programm findet nach Bereitstellung der Eingabedaten in dieser Sicht nicht statt.

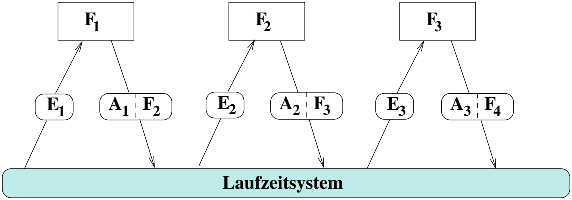
Diese Sicht

- 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

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 16, Ein-/Ausgabe):



Peter Pepper. *Funktionale Programmierung*.
Springer–Verlag, 2003, S.253

Naheliegende Frage

Sind Aufgaben von

- Betriebssystemen
- Graphischen Benutzerschnittstellen
- (Eingebetteten) Steuerungssystemen
- Nebenläufigen Systemen, Web-Services, Internet

oder Problemstellungen wie

- Fahrzeuge autonom ihren Weg im realen Straßenverkehr zu vorgegebenen Zielen finden zu lassen
- ...

durch den funktionsorientierten Begriff intuitiver Berechenbarkeit gedeckt, d.h. allein algorithmisch mit einmaliger Eingabedatenbereitstellung ohne weitere Interaktion vorstellbar und beschreibbar?

Naheliegende Frage (fgs.)

...oder sind dies qualitativ andere Probleme nicht funktionaler Art?

Im Fall von Betriebssystemen:

- Endliche Berechnung/Verarbeitung? Terminierung?
- Welche Funktion wird berechnet?

Im Fall autonom verkehrender Fahrzeuge:

- Welche Funktion wird berechnet?
- Wie sehen Ein- und Ausgabe aus?

Im Fall von Webservices, Internet:

- Statisches System? Komponenten kommen dynamisch hinzu und verschwinden.

Für Aufgaben dieser Art scheint Interaktion unverzichtbar.

Intuitive Berechenbarkeit: Allgemein genug?

Ändert die Hinzunahme von Interaktion das Verständnis des Begriffs von Berechnung möglicherweise ähnlich grundlegend wie das Finden von Ackermann der heute nach ihm benannten Ackermann-Funktion das von intuitiver Berechenbarkeit?

Besonders angestoßen wurde diese Frage durch:

- Peter Wegner. Why Interaction is More Powerful Than Algorithms. Communications of the ACM 40(5):81-91, 1997.

Darunter liegt die tiefergehende Frage:

- Was ist Berechnung?

Reichen Antworten wie z.B. von

- Martin Davis. What is a Computation? Chapter in L.A. Steeb (Hrsg.), Mathematics Today – Twelve Informal Essays. Springer-V., 1978.

Intuitive Berechenbarkeit: Allgemein genug?

Im Sinn von Peter Wegner auf den Punkt gebracht:

Gilt die Church/Turing-These im schwachen Sinn:

- Wann immer es eine effektive mechanische Methode zur Berechnung einer (mathematischen) Funktion gibt, d.h. intuitiv berechenbar ist, dann kann sie durch eine Turing-Maschine oder im λ-Kalkül berechnet werden.

...oder im starken Sinn:

- Eine Turing-Maschine kann alles berechnen oder im λ-Kalkül kann alles berechnet werden, was ein "Computer" berechnen kann; sie können alle Probleme lösen, die als Berechnung ausgedrückt werden können (über berechenbare Funktionen hinaus).

  Salopp: Ein Problem ist lösbar, wenn es eine Turing-Maschine oder einen λ-Ausdruck zu seiner Berechnung gibt.

Eine offene Frage

Dies ist seitdem eine “für” und “wider” diskutierte 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.

Interaktion: Neues 'Ackermann-Potential'? (4)

- 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.

Die Untersuchung geht weiter — eigenes Verständnis für und eigene Einsicht in Voraussetzungen und Implikationen der "Für"- und "Wider"-Argumente sind gefordert. Weitere Literaturhinweise hierzu sind in Anhang A angeführt.

Zurück zum λ-Kalkül

Der λ-Kalkül zeichnet sich aus durch:

- Einfachheit
  ...nur wenige syntaktische Konstrukte, einfache Semantik
- Ausdruckskraft
  ...Turing-mächtig, alle "intuitiv berechenbaren" Funktionen im λ-Kalkül ausdrückbar

Darüberhinaus:

- Bindeglied zwischen funktionalen Hochsprachen und ihren maschinennahen Implementierungen.

Reiner vs. angewandte λ-Kalküle

Wir unterscheiden:

- Reiner λ-Kalkül
  ...reduziert auf das "absolut Notwendige"
  $\leadsto$ besonders bedeutsam für Untersuchungen zur Theorie der Berechenbarkeit
- Angewandte λ-Kalküle
  ...syntaktisch angereichert, praxis- und programmiersprachennäher
- Extrem angereicherter angewandter λ-Kalkül
  ...funktionale Programmiersprache!

Kapitel 10.2

Syntax des λ-Kalküls

Syntax des (reinen) λ-Kalküls

Die Menge E der Ausdrücke des (reinen) λ-Kalküls, kurz λ-Ausdrücke, ist in folgender Weise definiert:

- Jeder Name (Identifikator) ist in E.
  Bsp: $a, b, c, \ldots, x, y, z, \ldots$
- Abstraktion: Wenn x ein Name und e aus E ist, dann ist auch $(\lambda x.\, e)$ in E.

  Sprechweise: Funktionsabstraktion mit formalem Parameter x und Rumpf e.
  Bsp.: $(\lambda x.(x\ x))$, $(\lambda x.(\lambda y.(\lambda z.(x\ (y\ z)))))$,...
- Applikation: Wenn f und e in E sind, dann ist auch (f e) in E.

  Sprechweisen: Anwendung von f auf e; f heißt auch Rator, e auch Rand.
  Bsp.: $((\lambda x.(x\ x))\ y)$,...

Syntax des (reinen) λ-Kalküls (fgs.)

Alternativ: Die Syntax in Backus-Naur-Form (BNF)

$$
\begin{array}{lll}
e & ::= & x \qquad\qquad \text{(Namen (Identifikatoren))} \\
 & ::= & \lambda\, x.e \qquad \text{(Abstraktion)} \\
 & ::= & e\ e \qquad\quad \text{(Applikation)} \\
 & ::= & (e) \qquad\quad \text{(Klammerung)}
\end{array}
$$

Vereinbarungen und Konventionen

- Überflüssige Klammern können weggelassen werden.

  Dabei gilt:

  - Rechtsassoziativität für λ-Sequenzen in Abstraktionen

    Beispiele:
    – $\lambda x.\lambda y.\lambda z.(x\,(y\,z))$ kurz für $(\lambda x.(\lambda y.(\lambda z.(x\,(y\,z)))))$
    – $\lambda x.\,e$ kurz für $(\lambda x.\,e)$

  - Linksassoziativität für Applikationssequenzen

    Beispiele:
    – $e_1\ e_2\ e_3 \ldots e_n$ kurz für $(\ldots((e_1\ e_2)\ e_3)\ldots e_n)$,
    – $(e_1\ e_2)$ kurz für $e_1\ e_2$

- Der Rumpf einer λ-Abstraktion ist der längstmögliche dem Punkt folgende λ-Ausdruck

  Beispiel:
  – $\lambda x.e\ f$ entspricht $\lambda x.(e\ f)$, nicht $(\lambda x.\ e)\ f$

Freie und gebundene Variablen (1)

...in λ-Ausdrücken:

Die Menge der

- freien Variablen:

$$\begin{aligned} \mathit{free}(\mathrm{x}) &= \{\mathrm{x}\}, \text{ wenn x ein Name/Identifikator ist} \\ \mathit{free}(\lambda \mathrm{x}.\mathrm{e}) &= \mathit{free}(\mathrm{e}) \setminus \{\mathrm{x}\} \\ \mathit{free}(\mathrm{f\ e}) &= \mathit{free}(\mathrm{f}) \cup \mathit{free}(\mathrm{e}) \end{aligned}$$

- gebundenen Variablen:

$$\begin{aligned} \mathit{bound}(\lambda \mathrm{x}.\mathrm{e}) &= \mathit{bound}(\mathrm{e}) \cup \{\mathrm{x}\} \\ \mathit{bound}(\mathrm{f\ e}) &= \mathit{bound}(\mathrm{f}) \cup \mathit{bound}(\mathrm{e}) \end{aligned}$$

Beachte: "gebunden" ist verschieden von "nicht frei"!
(anderenfalls wäre etwa "x gebunden in y")

Freie und gebundene Variablen (2)

Beispiel: Betrachte den λ-Ausdruck (λx. (x y)) x

- Im Gesamtausdruck (λx. (x y)) x :
  - x kommt frei und gebunden vor in (λx. (x y)) x
  - y kommt frei vor in (λx. (x y)) x
- In den Teilausdrücken von (λx. (x y)) x :
  - x kommt gebunden vor in (λx. (x y)) und frei in (x y) und x
  - y kommt frei vor in (λx. (x y)), (x y) und y

Gebunden vs. gebunden an

Wir müssen unterscheiden:

- Eine Variable ist gebunden
- Ein Variablenvorkommen ist gebunden an

Gebunden und gebunden an sind unterschiedliche Konzepte!

Letzteres meint:

- Ein (definierendes oder angewandtes) Variablenvorkommen ist an ein definierendes Variablenvorkommen gebunden

Festlegung

- Definierendes Variablenvorkommen: Vorkommen unmittelbar nach einem λ
- Angewandtes Variablenvorkommen: Jedes nicht definierende Variablenvorkommen

Kapitel 10.3

Semantik des λ-Kalküls

Semantik des (reinen) λ-Kalküls

Zentral für die Festlegung der Semantik sind folgende Hilfsbegriffe:

- Syntaktische Substitution
- Konversionsregeln / Reduktionsregeln

Syntaktische Substitution (1)

...ist 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'.

Informell:

- Der Ausdruck

  e' [e/x]

  bezeichnet denjenigen Ausdruck, der aus e' entsteht, indem jedes freie Vorkommen von x in e' durch e ersetzt, substituiert wird.

Beachte: Die obige informelle Beschreibung nimmt keinen Bedacht auf mögliche Bindungsfehler. Das leistet erst der wie folgt formal festgelegte Begriff syntaktischer Substitution.

Syntaktische Substitution (2)

Formale Definition der syntaktischen Substitution:

$$\cdot[\cdot/\cdot] : E \to E \to V \to E$$

$$\mathsf{x}\,[\mathsf{e}/\mathsf{x}] = \mathsf{e}, \text{ wenn x ein Name ist}$$

$$\mathsf{y}\,[\mathsf{e}/\mathsf{x}] = \mathsf{y}, \text{ wenn y ein Name mit } \mathsf{x} \neq \mathsf{y} \text{ ist}$$

$$(\mathsf{f\ g})\,[\mathsf{e}/\mathsf{x}] = (\mathsf{f}\,[\mathsf{e}/\mathsf{x}])\ (\mathsf{g}\,[\mathsf{e}/\mathsf{x}])$$

$$(\lambda \mathsf{x.f})\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{x.f}$$

$$(\lambda \mathsf{y.f})\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{y}.(\mathsf{f}\,[\mathsf{e}/\mathsf{x}]), \text{ wenn } \mathsf{x} \neq \mathsf{y} \text{ und } \mathsf{y} \notin \mathit{free}(\mathsf{e})$$

$$(\lambda \mathsf{y.f})\,[\mathsf{e}/\mathsf{x}] = \lambda \mathsf{z}.((\mathsf{f}\,[\mathsf{z}/\mathsf{y}])\,[\mathsf{e}/\mathsf{x}]), \text{ wenn } \mathsf{x} \neq \mathsf{y} \text{ und } \mathsf{y} \in \mathit{free}(\mathsf{e}),$$
$$\text{wobei z neue Variable mit } \mathsf{z} \notin \mathit{free}(\mathsf{e}) \cup \mathit{free}(\mathsf{f})$$

Syntaktische Substitution (3)

Illustrierende Beispiele:

- $((x\ y)\ (y\ z))\ [(a\ b)/y] = ((x\ (a\ b))\ ((a\ b)\ z))$
- $\lambda x.\ (x\ y)\ [(a\ b)/y] = \lambda x.\ (x\ (a\ b))$
- $\lambda x.\ (x\ y)\ [(a\ b)/x] = \lambda x.\ (x\ y)$
- Achtung: $\lambda x.\ (x\ y)\ [(x\ b)/y] \rightsquigarrow \lambda x.\ (x\ (x\ b))$

  $\rightsquigarrow$ ohne Umbenennung Bindungsfehler!
  ("x wird eingefangen")

Deshalb: $\lambda x.\ (x\ y)\ [(x\ b)/y] = \lambda z.\ ((x\ y)[z/x])\ [(x\ b)/y]$
$= \lambda z.\ (z\ y)\ [(x\ b)/y]$
$= \lambda z.\ (z\ (x\ b))$

$\rightsquigarrow$ mit Umbenennung kein Bindungsfehler!

Konversionsregeln, λ-Konversionen

...der zweite grundlegende Begriff:

- α-Konversion (Umbenennung formaler Parameter)

  $$\lambda x.e \Leftrightarrow \lambda y.e[y/x], \textit{ wobei } y \notin \textit{free}(e)$$

- β-Konversion (Funktionsanwendung)

  $$(\lambda x.f)\ e \Leftrightarrow f[e/x]$$

- η-Konversion (Elimination redundanter Funktion)

  $$\lambda x.(e\ x) \Leftrightarrow e, \textit{ wobei } x \notin \textit{free}(e)$$

$\rightsquigarrow$ führen auf eine operationelle Semantik des λ-Kalküls.

Sprechweisen

...im Zusammenhang mit Konversionsregeln:

- Von links nach rechts gerichtete Anwendungen der β- und η-Konversion heißen β- und η-Reduktion.
- Von rechts nach links gerichtete Anwendungen der β-Konversion heißen β-Abstraktion.

Intuition hinter den Konversionsregeln

Noch einmal zusammengefasst:

- α-Konversion: Erlaubt die konsistente Umbenennung formaler Parameter von λ-Abstraktionen
- β-Konversion: Erlaubt die Anwendung einer λ-Abstraktion auf ein Argument
  (Achtung: Gefahr von Bindungsfehlern! Abhilfe: α-Konversion!)
- η-Konversion: Erlaubt die Elimination redundanter λ-Abstraktionen

Beispiel: $(\lambda x.\lambda y.x\ y)\ (y\ z) \Rightarrow \lambda y.((y\ z)\ y)$
$\rightsquigarrow$ ohne Umbenennung Bindungsfehler
("y wird eingefangen")

Beispiele zur λ-Reduktion

Beispiel 1:

$((\lambda func.\lambda arg.(func\ arg)\ \lambda x.x)\ \lambda s.(s\ s))$

$$
\begin{array}{lcl}
(\beta\text{-Reduktion}) & \Rightarrow & \lambda arg.(\lambda x.x\ arg)\ \lambda s.(s\ s) \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda x.x\ \lambda s.(s\ s)) \\
(\beta\text{-Reduktion}) & \Rightarrow & \lambda s.(s\ s)
\end{array}
$$

(Fertig: Keine weiteren β-, η-Reduktionen mehr anwendbar)

Beispiel 2:

$(\lambda x.\lambda y.x\ y)\ ((\lambda x.\lambda y.x\ y)\ a\ b)\ c$

$$
\begin{array}{lcl}
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda x.\lambda y.x\ y)\ ((\lambda y.a\ y)\ b)\ c \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda y.((\lambda y.a\ y)\ b)\ y)\ c \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda y.(a\ b)\ y)\ c \\
(\beta\text{-Reduktion}) & \Rightarrow & (a\ b)\ c
\end{array}
$$

(Fertig: Keine weiteren β-, η-Reduktionen mehr anwendbar)

Reduktionsfolgen und Normalformen (1)

- Ein λ-Ausdruck ist in Normalform, wenn er durch β-Reduktion und η-Reduktion nicht weiter reduzierbar ist.
- (Praktisch relevante) Reduktionsstrategien
  - Normale Ordnung (leftmost-outermost)
  - Applikative Ordnung (leftmost-innermost)

Reductionsfolgen und Normalformen (2)

Beachte:

- Nicht jeder λ-Ausdruck ist zu einem λ-Ausdruck in Normalform konvertierbar.

  Beispiel:

  (1) $\lambda x.(x\ x)\ \lambda x.(x\ x) \;\Rightarrow\; \lambda x.(x\ x)\ \lambda x.(x\ x) \;\Rightarrow\; \ldots$
  (hat keine Normalform: Endlosselbstreproduktion!)

  (2) $(\lambda x.y)\ (\lambda x.(x\ x)\ \lambda x.(x\ x)) \;\Rightarrow\; y$
  (hat Normalform!)

Reduktionsfolgen und Normalformen (3)

Hauptresultate:

- Wenn ein λ-Ausdruck zu einem λ-Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des λ-Ausdrucks zum (bis auf α-Konversion) selben λ-Ausdruck in Normalform.
- Durch Reduktionen im λ-Kalkül sind genau jene Funktionen berechenbar, die Turing-, Markov-, μ-rekursiv, etc., berechenbar sind (und umgekehrt)!

Church-Rosser-Theoreme

Seien e_1 und e_2 zwei λ-Ausdrücke:

Theorem (Konfluenz-, Diamanteigenschaftstheorem)

Wenn $e_1 \Leftrightarrow e_2$, dann gibt es einen λ-Ausdruck e mit $e_1 \Rightarrow^* e$ und $e_2 \Rightarrow^* e$

Informell: Wenn eine Normalform ex., dann ist sie (bis auf α-Konversion) eindeutig bestimmt!

Theorem (Standardisierungstheorem)

Wenn $e_1 \Rightarrow^* e_2$ und e_2 in Normalform, dann gibt es eine normale Reduktionsfolge von e_1 nach e_2

Informell: Normale Reduktion terminiert am häufigsten, d.h. so oft wie überhaupt möglich!

Church-Rosser-Theoreme (fgs.)

Die Church-Rosser-Theoreme implizieren:

- λ-Ausdrücke in Normalform lassen sich (abgesehen von α-Konversionen) nicht weiter reduzieren, vereinfachen.
- Das 1. Church-Rosser-Theorem garantiert, dass die Normalform eines λ-Ausdrucks (bis auf α-Konversionen) eindeutig bestimmt ist, wenn sie existiert.
- Das 2. Church-Rosser-Theorem garantiert, dass eine normale Reduktionsordnung mit der Normalform terminiert, wenn es irgendeine Reduktionsfolge mit dieser Eigenschaft gibt.

Semantik des reinen λ-Kalküls

Die Church-Rosser-Theoreme und ihre Garantien erlauben die folgende Festlegung der Semantik des (reinen) λ-Kalküls:

- Die Semantik (Bedeutung) eines λ-Ausdrucks ist seine (bis auf α-Konversionen eindeutig bestimmte) Normalform, wenn sie existiert; die Normalform ist dabei der Wert des Ausdrucks.
- Existiert keine Normalform des λ-Ausdrucks, ist seine Semantik undefiniert.

Behandlung von Rekursion im reinen λ-Kalkül

Betrachte:

$$\text{fac n} = \text{if n} == 0 \text{ then } 1 \text{ else n} * \text{fac (n} - 1)$$

bzw. alternativ:

$$\text{fac} = \lambda\text{n.if n} == 0 \text{ then } 1 \text{ else n} * \text{fac (n} - 1)$$

Problem im reinen λ-Kalkül:

- λ-Abstraktionen (des reinen λ-Kalküls) sind anonym und können daher nicht (rekursiv) aufgerufen werden.
- Rekursive Aufrufe wie oben für die Funktion `fac` erforderlich können deshalb nicht naiv realisiert werden.

Kunstgriff: Der Y-Kombinator

Kombinatoren

- sind spezielle λ-Terme, λ-Terme ohne freie Variablen.

Y-Kombinator:

- $\mathsf{Y} = \lambda \mathsf{f}.(\lambda \mathsf{x}.(\mathsf{f}\ (\mathsf{x}\ \mathsf{x}))\ \lambda \mathsf{x}.(\mathsf{f}\ (\mathsf{x}\ \mathsf{x})))$

Zentrale Eigenschaft des Y-Kombinators:

- Für jeden λ-Ausdruck e ist (Y e) zu (e (Y e)) konvertierbar:

$$
\begin{array}{rcl}
Y\ e & \Leftrightarrow & (\lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x))))\ e \\
 & \Rightarrow & \lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x)) \\
 & \Rightarrow & e\ (\lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x))) \\
 & \Leftrightarrow & e\ (Y\ e)
\end{array}
$$

Kunstgriff: Der Y-Kombinator (fgs.)

Mithilfe des Y-Kombinators lässt sich Rekursion realisieren:

- Rekursion wird dabei auf Kopieren zurückgeführt

  Idee:

  ...überführe eine rekursive Darstellung in eine nicht-rekursive Darstellung, die den Y-Kombinator verwendet:

$$
\begin{array}{lr}
\quad \mathsf{f} = \cdots \ \mathsf{f} \cdots & \text{(rekursive Darstellung)} \\
\rightsquigarrow \mathsf{f} = \lambda \mathsf{f}.(\cdots \ \mathsf{f} \cdots)\ \mathsf{f} & \text{(λ-Abstraktion)} \\
\rightsquigarrow \mathsf{f} = \mathsf{Y}\ \lambda \mathsf{f}.(\cdots \ \mathsf{f} \cdots) & \text{(nicht-rekursive Darstellung)}
\end{array}
$$

  Bemerkung:

  - Vergleiche den Effekt des Y-Kombinators mit der Kopierregelsemantik prozeduraler Programmiersprachen.

Anwendung des Y-Kombinators

Zur Übung: Betrachte

$$\text{fac} = \text{Y}\ \lambda f.(\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * f\ (n-1))$$

Rechne nach:

$$\text{fac}\ 1 \Rightarrow\ \ldots\ \Rightarrow 1$$

Überprüfe bei der Rechnung:

- Der Y-Kombinator realisiert Rekursion durch wiederholtes Kopieren

Angewandte λ-Kalküle

...sind syntaktisch angereicherte Varianten des reinen λ-Kalküls.

Zum Beispiel:

- Konstanten, Funktionsnamen oder "übliche" Operatoren können Namen (im weiteren Sinn) sein
(Bsp: 1, 3.14, *true*, *false*, $+$, $*$, $-$, fac, simple,...)
- Ausdrücke können
  - komplexer sein
  (Bsp.: if e then e_1 else e_2 fi ...statt cond e e_1 e_2 für geeignet festgelegte Funktion cond)
  - getypt sein
  (Bsp.: 1 : *IN*, 3.14 : *IR*, *true* : *Boole*,...)
- ...

Angewandte λ-Kalküle (fgs.)

λ-Ausdrücke angewandter λ-Kalküle sind dann beispielsweise auch:

- Applikationen: fac 3, fib $(2+3)$, simple $x\ y\ z$ (entspricht $(((\text{simple}\ x)\ y)\ z)$),...
- Abstraktionen: $\lambda x.(x+x)$, $\lambda x.\lambda y.\lambda z.(x\ *\ (y\ -\ z))$, $2+3$, $(\lambda x.\text{if odd } x \text{ then } x*2 \text{ else } x \text{ div } 2 \text{ fi})\ 42, \ldots)$

Für das Folgende

...erlauben wir uns deshalb die Annehmlichkeit, Ausdrücke, für die wir eine eingeführte Schreibweise haben (z.B. $n * fac\ (n-1)$), in dieser gewohnten Weise zu schreiben.

Rechtfertigung:

- Resultate aus der theoretischen Informatik, insbesondere die Arbeit von

  Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941

  ...zur Modellierung von ganzen Zahlen, Wahrheitswerten, etc. durch (geeignete) Ausdrücke des reinen λ-Kalküls

Beispiel zur λ-Reduktion in angew. λ-Kalkülen

$$
\begin{array}{lcl}
\multicolumn{3}{l}{(\lambda x.\lambda y.x * y)\ ((\lambda x.\lambda y.x + y)\ 9\ 5)\ 3} \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda x.\lambda y.x * y)\ ((\lambda y.9 + y)\ 5)\ 3 \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda y.((\lambda y.9 + y)\ 5) * y)\ 3 \\
(\beta\text{-Reduktion}) & \Rightarrow & (\lambda y.(9 + 5) * y)\ 3 \\
(\beta\text{-Reduktion}) & \Rightarrow & (9 + 5) * 3
\end{array}
$$

(Verklemmt: Keine β-, η-Reduktionen mehr anwendbar)

- Weitere Regeln zur Reduktion primitiver Operationen in erweiterten λ-Kalkülen (Auswertung arithmetischer Ausdrücke, bedingte Anweisungen, Listenoperationen,...), sog. δ-Regeln.

$$
\begin{array}{lcl}
 & & (9 + 5) * 3 \\
(\delta\text{-Reduktion}) & \Rightarrow & 14 * 3 \\
(\delta\text{-Reduktion}) & \Rightarrow & 42
\end{array}
$$

Bemerkung

- Erweiterungen wie im vorigen Beispiel sind aus praktischer Hinsicht notwendig und einsichtig.
- Für theoretische Untersuchungen zur Berechenbarkeit (Theorie der Berechenkarkeit) sind sie kaum relevant.

Typisierte λ-Kalküle

...in typisierten λ-Kalkülen ist jedem λ-Ausdruck ein Typ zugeordnet.

Beispiele:

$$\begin{aligned} 3 &:: \text{Integer} \\ (*) &:: \text{Integer} \to \text{Integer} \to \text{Integer} \\ (\lambda x.2 * x) &:: \text{Integer} \to \text{Integer} \\ (\lambda x.2 * x)\ 3 &:: \text{Integer} \end{aligned}$$

Randbedingung: Typen müssen konsistent (wohlgetypt, wohltypisiert) sein.

Typisierte λ-Kalküle (fgs.)

...die Randbedingung induziert ein neues Problem im Zusammenhang mit Rekursion:

- Selbstanwendung im Y-Kombinator

$$\mathsf{Y} = \lambda \mathsf{f}.(\lambda \mathsf{x}.(\mathsf{f}\ (\mathsf{x}\ \mathsf{x}))\ \lambda \mathsf{x}.(\mathsf{f}\ (\mathsf{x}\ \mathsf{x})))$$

$\leadsto$ Y nicht endlich typisierbar!

(Eine pragmatische) Abhilfe:

- Explizite Rekursion zum Kalkül hinzufügen mittels Hinzunahme der Reduktionsregel $\mathsf{Y}\ \mathsf{e} \Rightarrow \mathsf{e}\ (\mathsf{Y}\ \mathsf{e})$

Bemerkung: Diese Hinzunahme ist zweckmäßig auch aus Effizienzgründen!

Resümee

Zurück zu Haskell:

- Haskell beruht auf typisiertem λ-Kalkül.
- Übersetzer, Interpretierer prüft, ob die Typisierung konsistent, wohlgetypt ist.
- Programmierer kann Typdeklarationen angeben (Sicherheit, aussagekräftigere Fehlermeldungen), muss aber nicht (bequem; manchmal jedoch unerwartete Ergebnisse, etwa bei zufällig korrekter, aber ungeplanter Typisierung (geplante Typisierung wäre inkonsistent gewesen und bei Angabe bei der Typprüfung als fehlerhaft aufgefallen)).
- Fehlende Typinformation wird vom Übersetzer, Interpretierer berechnet (inferiert).
- Rekursive Funktionen direkt verwendbar (für Haskell also kein Y-Kombinator erforderlich).

Zum Abschluss dieses Kapitels: Folgende Anekdote

...λ-artige Funktionsnotation in Haskell

...am Beispiel der Fakultätsfunktion:

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

Mithin in Haskell: "\" statt "λ" und "->" statt "."

Anekdote (vgl. P. Pepper "Funktionale Programmierung in Opal..."):

$$(\widehat{n}.n+1) \rightsquigarrow (\wedge n.n+1) \rightsquigarrow (\lambda n.n+1) \rightsquigarrow \backslash n\text{-}{>}n+1$$

Kapitel 10.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 10 (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 10 (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 10 (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 10 (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 10 (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 10 (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)
- William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 10 (7)

- 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 10 (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)

Teil V

Ergänzungen und weiterführende Konzepte

Kapitel 11

Muster und mehr

Muster, Komprehensionen und mehr

Muster und Musterpassung für

- elementare Datentypen
- Tupel
- Listen
  - `[]`-Muster
  - `(p:ps)`-Muster, auch als `(p:(q:qs))`-Muster, etc.
  - "as"-Muster
- algebraische Datentypen

Komprehensionen auf

- Listen
- Zeichenreihen

Listenkonstrukturen vs. Listenoperatoren

- Begriffsbestimmung und Vergleich

Muster und Musterpassung

- Muster sind (syntaktische) Ausdrücke
- Musterpassung (engl. pattern matching) erlaubt in Funktionsdefinitionen mithilfe einer Folge von Mustern aus einer Folge von Werten desselben Typs Alternativen auszuwählen; passt ein Wert auf ein Muster, wird diese Alternative ausgewählt.

Kapitel 11.1

Muster für elementare Datentypen

Muster und Musterpassung für elementare Datentypen (1)

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

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

or :: Bool -> Bool -> Bool
or False False = False
or _ _         = True

xor :: Bool -> Bool -> Bool
xor a b = a /= b
```

Muster und Musterpassung für elementare Datentypen (2)

```
add :: Int -> Int -> Int
add m 0 = m
add 0 n = n
add m n = m + n

mult :: Int -> Int -> Int
mult m 1 = m
mult 1 n = n
mult _ 0 = 0
mult 0 _ = 0
mult m n = m * n
```

Muster und Musterpassung für elementare Datentypen (3)

```
pow :: Integer -> Integer -> Integer
pow _ 0 = 1
pow m n = m * pow m (n-1)

sign :: Integer -> Integer
sign x
 | x > 0  = 1
 | x == 0 = 0
 | x < 0  = -1

ite :: Bool -> a -> a -> a
ite c t e = case c of True  -> t
                      False -> e
```

Muster und Musterpassung für elementare Datentypen (4)

```
conc :: String -> String -> String
conc "" t = t
conc s "" = s
conc s t  = s ++ t

doubleOrDelete :: Char -> String -> String
doubleOrDelete c s
 | c == 'D'
     = (head s) : ((head s) :
          doubleOrDelete c (tail s))  -- Verdoppeln
 | c == 'X'  = doubleOrDelete c (tail s)
                                      -- Löschen
 | otherwise = (head s) : doubleOrDelete c (tail s)
                                      -- Nichts
```

Muster und Musterpassung für elementare Datentypen (5)

Muster für elementare Datentypen sind:

- Konstanten elementarer Datentypen: `0`, `3.14`, `'c'`, `True`, `"aeiou"`,...
  $\rightsquigarrow$ ein Argument passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `m`, `n`,...
  $\rightsquigarrow$jedes Argument passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  $\rightsquigarrow$ jedes Argument passt (aber ist rechtsseitig nicht verwendbar).

Kapitel 11.2

Muster für Tupeltypen

Muster und Musterpassung für Tupeltypen (1)

```
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)
```

Muster und Musterpassung für Tupeltypen (2)

Muster für Tupeltypen sind:

- Konstanten von Tupeltypen: `(0,0)`, `(0,"Null")`, `(3.14,"pi",True)`,...
  ⇝ ein Argument passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `t`, `t1`,...
  ⇝jedes Argument passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  ⇝ jedes Argument passt (aber ist rechtsseitig nicht verwendbar).
- Kombinationen aus Konstanten, Variablen, Jokern:
  `(m,n)`, `(True,n,_)`, `(_,(m,_,n),3.14,k,_)`

Kapitel 11.3

Muster für Listen

Muster und Musterpassung für Listen (1)

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

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

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

null :: [a] -> Bool
null []    = True
null (_:_) = False
```

Muster und Musterpassung für Listen (3)

```
take :: Integer -> [a] -> [a]
take m ys = case (m,ys) of
              (0,_)    -> []
              (_,[])   -> []
              (n,x:xs) -> x : take (n - 1) xs

drop :: Integer -> [a] -> [a]
drop m ys =  case (m,ys) of
              (0,_)    -> ys
              (_,[])   -> []
              (n,_:xs) -> drop (n - 1) xs
```

Muster und Musterpassung für Listen (4)

Die Verwendung des Listenmusters `(t:ts)` ermöglicht eine einfachere Definition der Funktion `doubleOrDelete`:

```
doubleOrDelete :: Char -> String -> String
doubleOrDelete c (t:ts)
 | c == 'D'
    = t : (t : doubleOrDelete c ts)   -- Verdoppeln
 | c == 'X'  = doubleOrDelete c ts    -- Löschen
 | otherwise = t : doubleOrDelete c ts -- Nichts
```

Muster und Musterpassung für Listen (5)

Listenmuster erlauben auch, "tiefer" in eine Liste hineinzusehen:

```
maxElem :: Ord a => [a] -> a
maxElem []      = error "maxElem: Ungueltige Eingabe"
maxElem (y:[]) = y
maxElem (x:y:ys) = maxElem ((max x y) : ys)
```

Muster und Musterpassung für Listen (6)

Muster für Listen sind:

- Konstanten von Listentypen: `[]`, `[1,2,3]`, `[1..50]`, `[True,False,True,False]`,`['a'..'z']`,...
  $\leadsto$ ein Argument passt auf das Muster, wenn es eine Konstante vom entsprechenden Wert ist.
- Variablen: `p`, `q`,...
  $\leadsto$jedes Argument passt (und ist rechtsseitig verwendbar).
- Joker (eng. wild card): `_`
  $\leadsto$ jedes Argument passt (aber ist rechtsseitig nicht verwendbar).
- Konstruktormuster: `(p:ps)`, `(p:q:qs)`
  $\leadsto$ eine Liste L passt auf `(p:ps)`, wenn L nicht leer ist und der Kopf von L auf `p`, der Rest von L auf `ps` passt.

  Hinweis: Zur Passung auf `(p:ps)` reicht, dass L nicht leer ist.

Kapitel 11.4

Muster für algebraische Datentypen

Muster und Musterpassung für algebraische Typen (1)

```
data Jahreszeiten = Fruehling | Sommer
                  | Herbst | Winter
```

```
wetter :: Jahreszeiten -> String
wetter Fruehling = "Launisch"
wetter Sommer    = "Sonnig"
wetter Herbst    = "Windig"
wetter Winter    = "Frostig"
```

Muster und Musterpassung für algebraische Typen (2)

```
data Expr = Opd Int
          | Add Expr Expr
          | Sub Expr Expr
          | Squ Expr
```

```
eval :: Expr -> Int
eval (Opd n)      = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
eval (Squ e)      = (eval e)^2
```

Muster und Musterpassung für algebraische Typen (3)

```
data Tree a b = Leaf a
              | Node b (Tree a b) (Tree a b)

depth  :: (Tree a b) -> Int
depth (Leaf _)     = 1
depth (Node _ l r) = 1 + max (depth l) (depth r)


data List a = Empty | Head a (List a)

lgthList :: List a -> Int
lgthList Empty        = 0
lgthList (Head _ hs) = 1 + lgthList hs
```

Muster und Musterpassung für algebraische Typen (4)

Ähnlich wie für Listen, erlauben Muster auch, in algebraische Typen "tiefer" hineinzusehen:

```
data Tree = Leaf Int
          | Node Int Tree Tree

f :: Tree -> Int
f (Leaf _)      = 0
f (Node n (Leaf m) (Node p (Leaf q) (Leaf r)))
                = n+m+p+q+r
f (Node _ _ _) = 0
```

Muster und Musterpassung für algebraische Typen (5)

Muster für algebraische Typen sind:

- ...
- Konstruktoren:
  `Sommer`,
  `Winter`,
  `Opd e`,
  `(Node _ l r)`,
  `Leaf a`,
  `Leaf _`,
  ...

Kapitel 11.5

Das as-Muster

Das as-Muster (1)

Sehr nützlich ist oft das sog. as-Muster (@ gelesen als "as"):

```
nonEmptysuffixes :: String -> [String]
nonEmptysuffixes s@(_:ys) = s : suffixes ys
nonEmptysuffixes _ = []

nonEmptySuffixes "Curry"
   ->> ["Curry","urry","rry","ry","y"]
```

Bedeutung:

- `xs@(_:ys)`: Binde `xs` an den Wert, der auf die rechte Seite des `@`-Symbols passt.

Vorteile:

- `xs@(_:ys)` passt mit denselben Listenwerten zusammen wie `(_:ys)`; zusätzlich erlaubt es auf die Gesamtliste mittels `xs` Bezug zu nehmen statt (nur) mit `(_:ys)`.
- I.a. führt dies zu einfacheren und übersichtlicheren Definitionen.

Das as-Muster (2)

Zum Vergleich: Die Funktion nonEmptysuffixes

- mit as-Muster:

```
nonEmptysuffixes :: String -> [String]
nonEmptysuffixes s@(_:ys) = s : suffixes ys
nonEmptysuffixes _ = []
```

- ohne as-Muster:

```
nonEmptysuffixes :: String -> [String]
nonEmptysuffixes (y:ys) = (y:ys) : suffixes ys
nonEmptysuffixes _ = []
```

...weniger elegant und weniger gut lesbar.

Das as-Muster (3)

Listen und as-Muster:

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : l) ++ xs
```

Zum Vergleich wieder ohne as-Muster:

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

Das as-Muster (4)

Tupel und as-Muster:

```
swap :: (a,a) -> (a,a)
swap p@(c,d)
 | c /= d = (d,c)
 | otherwise = p
```

Zum Vergleich ohne as-Muster:

```
swap :: (a,a) -> (a,a)
swap (c,d)
 | c /= d = (d,c)
 | otherwise = (c,d)
```

Das as-Muster (5)

Tupel und as-Muster:

```
triswap :: (a,Bool,a) -> (a,Bool,a)
triswap t@(b,c,d)
 | c     = (d,c,b)
 | not c = t
```

Zum Vergleich ohne as-Muster:

```
triswap :: (a,Bool,a) -> (a,Bool,a)
triswap (b,c,d)
 | c     = (d,c,b)
 | not c = (b,c,d)
```

Resümee

Musterbasierte Funktionsdefinitionen

- sind elegant
- führen (i.a.) zu knappen, gut lesbaren Spezifikationen.

Zur Illustration: Die Funktion `binom` mit Mustern; und 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 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))
```

Resümee (fgs.)

Aber:

Musterbasierte Funktionsdefinitionen können auch

- zu subtilen Fehlern führen
- Programmänderungen/-weiterentwicklungen erschweren, "bis hin zur Tortur", etwa beim Hinzukommen eines oder mehrerer weiterer Parameter

  (siehe S. 164 in: Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003.)

Kapitel 11.6

Komprehensionen

Komprehensionen

...ein für funktionale Programmiersprachen

- charakteristisches
- elegantes und ausdruckskräftiges Ausdrucksmittel

Komprehensionen auf:

- Listen
- Zeichenreihen (spezielle Listen)

Listenkomprehension in Ausdrücken

```
lst1 = [1,2,3,4]

[3*n | n <- lst1] ->> [3,6,9,12]


lst2 = [1,2,4,7,8,11,12,42]

[ square n | n <- lst2 ]
    ->> [1,4,16,49,64,121,144,1764]

[ n | n <- lst2, isPowOfTwo n ] ->> [1,2,4,8]

[ n | n <- lst2, isPowOfTwo n, n>=5 ] ->> [8]

[ isPrime n | n <- lst2 ]
 ->> [False,True,False,True,False,True,False,False]
```

Listenkomprehension in Fkt.-Definitionen (1)

```
addCoordinates :: [Point] -> [Float]
addCoordinates pLst
       = [x+y | (x,y) <- pLst, (x>0 || y>0)]

addCoordinates [(0.0,0.5),(3.14,17.4),(-1.5,-2.3)]
       ->> [0.5,20.54]


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  c  = False

grabCapVowels "Auf Eine Informatik Ohne Verdruss"
               ->> "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 ]
```

Bemerkung: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters `x:xs` in `quickSort (x:xs) = ...`

Zeichenreihenkomprehensionen (1)

Zeichenreihen sind in Haskell ein Listen-Typalias:

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

Es gilt:

```
"Haskell" == ['H','a','s','k','e','l','l']
```

Daher stehen für Zeichenreihen dieselben

- Funktionen
- Komprehensionen

zur Verfügung wie für allgemeine Listen.

Zeichenreihenkomprehensionen (2)

Beispiele:

```
"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)]
```

Zeichenreihenkomprehensionen (3)

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, isLower x]

lowers "Haskell" ->> 6
```

```
count :: Char -> String -> Int
count c xs = length [x | x <- xs, x == c]

count ’s’ "Mississippi" ->> 4
```

Kapitel 11.7

Listenkonstruktoren, Listenoperatoren

Listenkonstruktoren vs. Listenoperatoren (1)

Der Operator

- (:) ist Listenkonstruktor
- (++) ist Listenoperator

Abgrenzung: Konstruktoren führen zu eindeutigen Darstellungen, gewöhnliche Operatoren nicht.

Beispiel:

```
42:17:4:[]  ==  (42:(17:(4:[]))) -- eindeutig

[42,17,4]   == [42,17] ++ [] ++ [4]
            == [42] ++ [17,4] ++ []
            == [42] ++ [] ++ [17,4]
            == ...
                                  -- nicht eindeutig
```

Listenkonstruktoren vs. Listenoperatoren (2)

Bemerkung:

- `(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.

Listenkonstruktoren vs. Listenoperatoren (3)

Wg. der fehlenden Zerlegungseindeutigkeit bei Verwendung von Listenoperatoren dürfen

- Listenoperatoren nicht in Mustern verwendet werden

Listenkonstruktoren vs. Listenoperatoren (4)

Beispiel:

```
cutTwo :: (Char,Char) -> String -> String
cutTwo _ ""          = ""
cutTwo _ (s:[])      = [s]
cutTwo (c,d) (s:(t:ts))
 | (c,d) == (s,t) = cutTwo (c,d) ts
 | otherwise      = s : cutTwo (c,d) (t:ts)
```

...ist syntaktisch korrekt.

```
cutTwo :: (Char,Char) -> String -> String
cutTwo _ ""          = ""
cutTwo _ (s:[])      = [s]
cutTwo (c,d) s@([s1]++[s2])
 | [c,d] == s1 = cutTwo (c,d) s2
 | otherwise   = head s : cutTwo (c,d) tail s
```

...ist syntaktisch inkorrekt.

Kapitel 11.8

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 11 (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, 2007. (Kapitel 4.4, Pattern matching; Kapitel 5, List comprehensions)

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 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 11 (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)

Kapitel 12

Module

Module

...unterstützen

- Programmieren im Großen (Kap. 12.1)

...gilt auch für

- Haskells Modulkonzept (Kap. 12.2)

...spezielle Anwendung

- Abstrakte Datentypen (Kap. 12.3)

Kapitel 12.1

Programmieren im Großen

Zur Modularisierung im Generellen (1)

Intuitiv:

- Zerlegung großer Programm(systeme) in kleinere Einheiten, genannt Module

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems

Zur Modularisierung im Generellen (2)

Vorteile:

- Arbeitsphysiologisch: Unterstützung arbeitsteiliger Programmierung
- Softwaretechnisch: Unterstützung der Wiederbenutzung von Programmen und Programmteilen
- Implementierungstechnisch: Unterstützung von getrennter Übersetzung (separate compilation)

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Steigerung der Qualität (Verlässlichkeit) und Reduzierung der Kosten

Zur Modularisierung im Generellen (3)

Anforderungen an Modulkonzepte 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?

  in Modulen.

Regeln “guter” Modularisierung (1)

- Aus (lokaler) Modulsicht:

  Module sollen:

  - einen klar definierten, auch unabhängig von anderen Modulen verständlichen Zweck besitzen
  - nur einer Abstraktion entsprechen
  - einfach zu testen sein

- Aus (globaler) Gesamtsystemsicht:

  Aus Modulen aufgebaute Programme sollen so entworfen sein, dass

  - Auswirkungen von Designentscheidungen (z.B. Einfachheit vs. Effizienz einer Implementierung) auf wenige Module beschränkt sind
  - Abhängigkeiten von Hardware oder anderen Programmen auf wenige Module beschränkt sind

Regeln "guter" Modularisierung (2)

Zwei zentrale Konzepte in diesem Zusammenhang sind:

- Kohäsion: Intramodulare Perspektive
- Kopplung: Intermodulare Perspektive

Regeln "guter" Modularisierung (3)

Aus intramodularer Sicht: Kohäsion

- Anzustreben sind:
  - Funktionale Kohäsion (d.h. Funktionen ähnlicher Funktionalität sollten in einem Modul zusammengefasst sein, z.B. Ein-/Ausgabefunktionen, trigonometrische Funktionen, etc.)
  - Datenkohäsion (d.h. Funktionen, die auf den gleichen Datenstrukturen arbeiten, sollten in einem Modul zusammengefasst sein, z.B. Baummanipulationsfunktionen, Listenverarbeitungsfunktionen, etc.)

Regeln "guter" Modularisierung (4)

- Zu vermeiden sind:
  - Logische Kohäsion (d.h. unterschiedliche Implementierungen der gleichen Funktionalität sollten in verschiedenen Modulen untergebracht sein, z.B. verschiedene Benutzerschnittstellen eines Systems)
  - Zufällige Kohäsion (d.h. Funktionen sind ohne sachlichen Grund, zufällig eben, in einem Modul zusammengefasst)

Regeln "guter" Modularisierung (5)

Aus intermodularer Sicht: Kopplung

- beschäftigt sich mit dem Import-/Exportverhalten von Modulen
  - Anzustreben ist:
    - Datenkopplung (d.h. Funktionen unterschiedlicher Module kommunizieren nur durch Datenaustausch (in funktionalen Sprachen per se gegeben))

Regeln "guter" Modularisierung (6)

Kennzeichen "guter/gelungener" Modularisierung:

- Starke Kohäsion
  d.h. enger Zusammenhang der Definitionen eines Moduls
- Lockere Kopplung
  d.h. wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere weder direkte noch indirekte zirkuläre Abhängigkeiten.

Für eine weitergehende und vertiefende Diskussion siehe Kapitel 10 in: Manuel Chakravarty, Gabriele Keller. Einführung in die Programmierung mit Haskell, Pearson Studium, 2004.

Kapitel 12.2

Module in Haskell

Allgemeiner Aufbau eines Moduls in Haskell

```
module M where

-- Daten- und Typdefinitionen
data D_1 ... = ...
...
data D_n ... = ...

type T_1 = ...
...
type T_m = ...

-- Funktionsdefinitionen
f_1 :: ...
f_1 ... = ...
...
f_p :: ...
f_p ... = ...
```

Das Modulkonzept von Haskell

...unterstützt:

- Export
  - Selektiv/Nicht selektiv
- Import
  - Selektiv/Nicht selektiv
  - Qualifiziert
  - Mit Umbenennung

von Datentypen und Funktionen.

Nicht unterstützt:

- Automatischer Reexport!

Nicht selektiver Import

Generelles Muster:

```
module M1 where
...

module M2 where
import M1  -- Alle im Modul M1 (sichtbaren)
           -- Bezeichner/Definitionen werden
           -- importiert und können in M2
           -- benutzt werden.
           -- Somit: Nicht selektiver Import!
```

Selektiver Import

Generelles Muster:

```
module M1 where
...

module M2 where
import M1 (D_1 (..), D_2, T_1, f_5)
 -- Ausschließlich D_1 (einschließlich Konstrukto-
 -- ren), D_2 (ohne Konstruktoren), T_1 und f_5 wer-
 -- den importiert und können in M2 benutzt werden.
 -- Somit: Importiere nur, was explizit genannt ist!

module M3 where
import M1 hiding (D_1, T_2, f_1)
 -- Alle (sichtbaren) Bezeichner/Definitionen mit Aus-
 -- nahme der explizit genannten werden importiert und
 -- können in M3 benutzt werden.
 -- Somit: Importiere alles, was nicht explizit uner-
 -- wünscht ist und ausgeschlossen wird!
```

Anwendung

Erinnerung (vgl. Kapitel 1):

- "Verstecken" der in `Prelude.hs` vordefinierten Funktionen `reverse`, `tail` und `zip` durch Ergänzung der Zeile

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

  ...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden).

Nicht selektiver Export

Generelles Muster:

```
module M1 where       -- Alle im Modul M1 (sichtbaren)
                      -- Bezeichner/Definitionen werden
data D_1 ... = ...    -- exportiert und können von an-
...                   -- deren Modulen importiert werden.
data D_n ... = ...    -- Somit: Nicht selektiver Export!

type T_1 = ...
...
type T_m = ...

f_1 :: ...
f_1 ... = ...
...
f_p :: ...
f_p ... = ......
```

Selektiver Export

Generelles Muster:

```
module M1 (D_1 (..), D_2, T_1, f_2, f_5) where

data D_1 ... = ...  -- Nur die explizit genannten
...                 -- Bezeichner/Definitionen werden
data D_n ... = ...  -- exportiert und können von
                    -- anderen Modulen importiert
type T_1 = ...      -- werden.
...                 -- Unterstützt Geheimnisprinzip!
type T_m = ...      -- Somit: Selektiver Export!

f_1 :: ...
f_1 ... = ...
...
f_p :: ...
f_p ... = ......
```

Kein automatischer Reexport

Veranschaulichung:

```
module M1 where...

module M2 where
import M1 -- Nicht selektiver Import. Das heißt:
...       -- Alle im Modul M1 (sichtbaren) Bezeich-
f_2M2     -- ner/Definitionen werden importiert und
...       -- können in M2 benutzt werden.

module M3 where
import M2 -- Nicht selektiver Import. Aber: Zwar wer-
...       -- den alle im Modul M2 (sichtbaren) Bezeich-
          -- ner/Definitionen importiert und können
          -- in M3  benutzt werden, nicht jedoch die
          -- von M2 aus M1 importierten Namen.
          -- Somit: Kein automatischer Reexport!
```

Händischer Reexport

Abhilfe: Händischer Reexport!

```
module M2 (M1,f_2M2) where
import M1         -- Nicht selektiver Reexport:
...               -- M2 reexportiert jeden aus M1
f_2M2             -- importierten Namen
...

module M2 (D_1 (..), f_1, f_2) where
import M1         -- Selektiver Reexport:
...               -- M2 reexportiert von den aus M1
f_2M2             -- importierten Namen ausschließlich
...               -- D_1 (einschließlich Konstruktoren),
                  -- f_1 und f_2
```

Namenskonflikte, Umbenennungen

- Namenskonflikte
  - Auflösen der Konflikte durch qualifizierten Import
    `import qualified M1`
- Umbenennen importierter Module und Bezeichner
  - Lokale Namen importierter
    - Module
      `import MyM1 as M1`
    - Bezeichner
      `import M1 (f1,f2) renaming (f1 to g, f2 to h)`

Konventionen und 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 `module`.
- Gute Praxis
  - Module unterstützen eine (!) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium)
  - Module sind "kurz" (ideal: 2 bis 3 Bildschirmseiten; grundsätzlich: "so lang wie nötig, so kurz wie möglich")

Das Hauptmodul

Modul `main`

- ...muss in jedem Modulsystem als “top-level” Modul vorkommen und eine Funkion namens `main` festlegen.

  $\rightsquigarrow$ ...ist der in einem übersetzten System bei Ausführung des übersetzten Codes zur Auswertung kommende Ausdruck.

Kapitel 12.3

Abstrakte Datentypen

Anwendung des Modulkonzepts

...zur Definition abstrakter Datentypen, kurz: ADTs

Mit ADTs verfolgtes Ziel:

- Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. information hiding)

Implementierungstechnischer Schlüssel:

- Haskells Modulkonzept, speziell selektiver Export

Abstrakte Datentypen (1)

Grundlegende Idee:

Implizite Festlegung des Datentyp und seiner Werte in zwei Teilen:

- A) Schnittstellenfestlegung: Angabe der auf den Werten des Datentyps zur Verfügung stehenden Operationen durch ihre 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.

Abstrakte Datentypen (2)

ADT-Beispiel: FIFO-Warteschlange Queue

A: Festlegung der Schnittstelle durch Signaturangabe:

```
NEW:                  -> Queue
ENQUEUE:  Queue x 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)
```

Abstrakte Datentypen (3)

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 sind 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.

Abstrakte Datentypen (4)

Drei wegbereitende Arbeiten:

- 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.

Haskell-Realisierung des ADTs Queue (1)

```
-- Schnittstellenspezifikation:

module Queue (Queue,     -- Geheimnisprinzip:
                         -- Kein Konstruktorexport!
              new,       -- Queue a
              enqueue,   -- Queue a -> a -> Queue a
              front,     -- Queue a -> a
              dequeue,   -- Queue a -> Queue a
              is_empty,  -- Queue a -> Bool
              ) where
```

Haskell-Realisierung des ADTs Queue (2)

```
new      :: Queue a
enqueue  :: Queue a -> a -> Queue a
front    :: Queue a -> a
dequeue  :: Queue a -> Queue a
is_empty :: Queue a -> Bool

{- Gesetze in Form von Kommentaren:
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)
-}
```

Haskell-Realisierung des ADTs Queue (3)

```
-- Implementierung

data Queue = Qu [a]

new = Qu []

enqueue (Qu xs) x = Qu (xs ++ [x])

front q@(Qu xs)
  | not (is_empty q) = head xs
  | otherwise        = error "Nobody is waiting!"

is_empty (Qu []) = True
is_empty _       = False

dequeue q@(Qu xs)
  | not (is_empty q) = Qu (tail xs)
  | otherwise        = error "Nobody is waiting!"
```

Haskell-Realisierung des ADTs Queue (3)

Implementierungstechnische Anmerkung:

- Das "as"-Muster als nützliche notationelle Musterspielart:

```
front q@(Qu xs)   -- Arg. as "q" or as "(Qu xs)"
dequeue q@(Qu xs) -- Arg. as "q" or as "(Qu xs)"
```

  Das "as"-Muster `q@(Qu xs)` erlaubt mittels:

  - `q`: Zugriff auf das Argument als Ganzes
  - `(Qu xs)`: Zugriff auf/über die Struktur des Arguments

Haskell-Realisierung des ADTs Queue (4)

Konzeptuelle Anmerkungen:

- Haskell bietet kein dezidiertes Sprachkonstrukt zur Spezifikation von ADTs, das eine externe Offenlegung von Signaturen und Gesetzen bei intern bleibender Implementierung erlaubte.
- Der Behelf, ADTs mittels Modulen zu spezifieren, ermöglicht zwar die Implementierung intern und versteckt zu halten, allerdings können die Signaturen und Gesetze nur in Form von Kommentaren offengelegt werden.

Algebraische vs. abstrakte Datentypen

Ein Vergleich:

- Algebraische Datentypen
  - werden durch die Angabe ihrer Elemente spezifiziert, aus denen sie bestehen.
- Abstrakte Datentypen
  - werden durch ihr Verhalten spezifiziert, d.h. durch die Menge der Operationen, die darauf arbeiten.

Zusammenfassung

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.
  - Arbeitsteilige Programmierung.

Kapitel 12.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 12 (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 12 (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 12 (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 12 (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 13

Typüberprüfung, Typinferenz

Motivation und Erinnerung

Wir wissen bereits:

- Jeder gültige Haskell-Ausdruck hat einen definierten Typ; gültige Ausdrücke heißen wohlgetypt.
- Typen gültiger Haskell-Ausdrücke könnnen sein:
  - Monomorph
    `fac :: Integer -> Integer`
  - Polymorph
    - Uneingeschränkt polymorph
      `flip :: (a -> b -> c) -> (b -> a -> c)`
    - Eingeschränkt polymorph durch Typklassenbedingungen
      `elem :: Eq a => a -> [a] -> Bool`
- Der Typ eines Haskell-Ausdrucks kann
  - explizit ($\leadsto$ Typüberprüfung)
  - implizit ($\leadsto$ Typinferenz)

  im Programm angegeben sein.

Typisierte Programmiersprachen

Typisierte Programmiersprachen teilen sich in Sprachen mit

- schwacher ($\rightsquigarrow$ Typprüfung zur Laufzeit)
- starker ($\rightsquigarrow$ Typprüfung, -inferenz zur Übersetzungszeit)

Typisierung.

Haskell ist eine

- stark typisierte Programmiersprache.

Vorteile

...aus der Nutzung stark getypter Programmiersprachen:

- Verlässlicherer Code: Viele Programmierfehler können schon zur Übersetzungszeit entdeckt werden; der Nachweis der Typkorrektheit eines Programms ist ein Korrektheitsbeweis für dieses Programm auf dem Abstraktionsniveau von Typen.
- Effizienterer Code: Keine Typprüfungen zur Laufzeit erforderlich.
- Effektivere Programmentwicklung: Typinformation ist zusätzliche Programmdokumentation, die Verstehen, Wartung Pflege und Weiterentwicklung eines Programms vereinfacht, z.B. die Suche nach einer vordefinierten Bibliotheksfunktion ("Gibt es eine Bibliotheksfunktion, die alle Duplikate aus einer Liste entfernt, die angewendet auf die Liste `[2,3,2,1,3,4]` also das Resultat `[2,3,1,4]` liefert? Suche somit nach einer Funktion mit Typ `(Eq a) => [a] -> [a]`".)

Typüberprüfung und Typinferenz

...ist eine

- Schlüsselfähigkeit von Übersetzern und Interpretierern.

Dabei unterscheiden wir zwischen

- monomorpher
- polymorpher (parametrisch und *ad hoc* (Überladung))

Typüberprüfung und Typinferenz.

Beispiel (1)

Betrachte die Funktion magicType:

```
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)
```

Beispiel (2)

Automatische Typinferenz in Hugs mittels Aufrufs des Kommandos `:t magicType` 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
```

Beispiel (3)

Der Typ der Funktion `magicType` ist

- zweifellos komplex.

Das wirft die Frage auf:

- Wie gelingt es Übersetzern und Interpretierer Typen wie den der Funktion `magicType` automatisch zu inferieren?

Beispiel (4)

Informelle Antwort:

- Ausnutzung von Kontextinformation von Ausdrücken und Funktionsdefinitionen.

Die systematische Behandlung führt zu formaler Antwort:

Diese ist gekennzeichnet durch

- Typanalyse, Typüberprüfung
- Typsysteme
- Typinferenz

Wir beginnen mit einer beispielgetriebenen informellen Annäherung an diese Konzepte.

Typüberprüfung und Typinferenz

Generell ist zu unterscheiden zwischen

- monomorpher
- polymorpher

Typinferenz.

Die Grundidee ist in beiden Fällen dieselbe:

Beute

- die Kontextinformation des zu typisierenden Ausdrucks aus und ziehe daraus Rückschlüsse auf die beteiligten Typen.

Kapitel 13.1

Monomorphe Typüberprüfung

Monomorphe Typüberprüfung

Kennzeichnend für den Fall monomorpher Typüberprüfung ist:

- Ein Ausdruck ist
  - wohlgetypt, d.h. hat einen wohldefinierten eindeutig bestimmten Typ
  - nicht wohlgetypt, d.h. hat überhaupt keinen Typ

Vereinbarung für die folgenden Beispiele:

Die Polymorphie parametrisch oder überladen polymorpher Funktionen wird explizit durch geeignete Indizierung syntaktisch aufgelöst wie in

- $+_{Int}$ `:: Int -> Int -> Int`
- `length`$_{Char}$ `:: [Char] -> Int`

Typprüfung und -inferenz für Ausdrücke (1)

Im folgenden Beispiel erlaubt die

- Auswertung des Ausdruckskontexts korrekte Typung festzustellen.

Beispiel: Betrachte den Ausdruck `ord 'c' + 3`

```
Char -> Int   Char
      |        /
     \/      \/
      ord   'c'      +_Int       3
      /\            /\          /\
       |             |           |
      Int  (Int -> Int -> Int)  Int
```

- Analyse und Prüfung stellt korrekte Typung fest!

Typprüfung und -inferenz für Ausdrücke (2)

Im folgenden Beispiel erlaubt die

- Auswertung des Ausdruckskontexts inkorrekte Typung aufzudecken.

Beispiel: Betrachte den Ausdruck `ord 'c' + False`

```
           Char -> Int   Char
                |       /
               \/      \/
                ord   'c'      +_Int     False
                /\            /\          /\
                 |             |           |
                Int  (Int -> Int -> Int)  Bool
                /\    /\     /\           /\
                 |_____|      |____________|
                    |                |
Erwarteter und angegebener Typ   Erwartet: Int
stimmen überein: Typ-korrekt     Angegeben: Bool: Typ-Fehler
```

- Analyse und Prüfung deckt inkorrekte Typung auf!

Typprüfung monomorpher Fkt.-Definitionen

Für die Typprüfung monomorpher Funktionsdefinitionen

```
f :: t1 -> t2 -> ... -> tk -> t
f p1 p2 ... pk
  | b1 = e1
  | b2 = e2
  ...
  | bn = en
```

sind drei Eigenschaften zu überprüfen:

1. Jeder Wächter $\texttt{b}_i$ muss vom Typ `Bool` sein.
2. Jeder Ausdruck $\texttt{e}_i$ muss einen Wert vom Typ `t` haben.
3. Das Muster jedes Parameters $\texttt{p}_i$ muss konsistent mit dem entsprechenden Typ $\texttt{t}_i$ sein.

Konsistenz und Passung von Mustern

Informell:

Ein Muster ist konsistent mit einem Typ, wenn es auf einige oder alle Werte dieses Typs passt.

Im Detail:

- Eine Variable ist mit jedem Typ konsistent.
- Ein Literal oder Konstante ist mit ihrem Typ konsistent.
- Ein Muster `(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 13.2

Polymorphe Typüberprüfung

Polymorphe Typüberprüfung und Typinferenz

Kennzeichnend für den Fall polymorpher Typüberprüfung ist:

- Ein Ausdruck ist
  - wohlgetypt und kann mehrere wohldefinierte konkrete Typen haben
  - nicht wohlgetypt, d.h. hat überhaupt keinen Typ

Der Schlüssel zur algorithmischen Lösung des polymorphen Typüberprüfungsproblems ist

- Randbedingungserfüllung (engl. constraint satisfaction)

auf der Grundlage von

- Unifikation.

Veranschaulichung (1)

Betrachte die Funktionsdefinition

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

Informell steht der polymorphe Typ

```
[a] -> Int
```

abkürzend für die Menge monomorpher Typen

```
[t] -> Int
```

wobei t für einen beliebigen monomorphen Typ steht; insgesamt steht [a] -> Int also abkürzend für die (unendliche) Menge konkreter Typen

```
[Int] -> Int
[(Bool,Char)] -> Int
...
```

Veranschaulichung (2)

Im Beispiel des Aufrufs

```
length ['c','a']
```

können wir aus dem Aufrufkontext auf den konkreten Typ der Funktion `length` in diesem Kontext schließen:

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

Beobachtung

Die vorstehenden Beispiele erlauben uns die Schlussfolgerung:

Der Anwendungskontext von Ausdrücken legt

- implizit Randbedingungen an die Typen von Ausdrücken fest.

Auf diese Weise reduziert sich das Typüberprüfungsproblem auf die

- Bestimmung möglichst allgemeiner Typen für die verschiedenen Ausdrücke, so dass keine Randbedingung verletzt ist.

Polymorphe Typprüfung und Typinferenz (1)

Im folgenden Beispiel 1 erlaubt die

- Auswertung des Ausdruckskontexts den allgemeinsten Typ zu inferieren.

Beispiel 1: Betrachte den Funktionsaufruf `f e`

```
 f muss Funktionstyp haben      e muss vom Typ s sein

       s -> t                            s
           \________       ___________/
                     \   /
                     \/ \/
                      f e
                      /\
                       |

 f e muss (Resultat-) Typ t haben
```

- Typüberprüfung und -inferenz liefern, dass die allgemeinsten Typen der Ausdrücke e, f und f e sind: `e :: s`, `f :: s -> t` und `f e :: t`

Polymorphe Typprüfung und Typinferenz (2)

Beispiel 2: Betrachte die Funktionsgleichung:

```
f (x,y) = (x , ['a' .. y])
```

Beobachtung:

- f erwartet Paare als Argumente, wobei
  - 1-te Komponente: ohne weitere Randbedingung, also von irgendeinem Typ ist.
  - 2-te Komponente: y muss Typ Char haben, da y als Schranke eines Werts ['a' .. y] eines Aufzählungstyps benutzt wird.

Somit können wir für den Typ von f schließen:

```
f :: (a , Char) -> (a , [Char])
```

Polymorphe Typprüfung und Typinferenz (3)

Beispiel 3: Betrachte die Funktionsgleichung:

```
g (m,zs) = m + length zs
```

Beobachtung:

- `g` erwartet Paare als Argumente, wobei
  - 1-te Komponente: `m` muss einen numerischen Typ haben, da `m` als Operand von `+` verwendet wird.
  - 2-te Komponente: `zs` muss Typ `[b]` haben, da `zs` als Argument der Funktion `length` verwendet wird, die den Typ `[b] -> Int` hat.

Somit können wir für den Typ von `g` schließen:

```
g :: (Int, [b]) -> Int
```

Polymorphe Typprüfung und Typinferenz (4)

Beispiel 4: Betrachte die Komposition der Funktionen `g` und `f` der beiden vorangegangenen Beispiele:

```
g . f
```

Beobachtung:

- `g . f`: In einem komponierten Ausdruck `g . f` ist der Resultatwert von `f` der Argumentwert von `g`.

  Konkret für dieses Beispiel bedeutet dies:

  - Resultattyp von `f` ist `(a , [Char])`.
  - Argumenttyp von `g` ist `(Int , [b])`.
  - Gesucht: Typinstanzen für die Typvariablen `a` und `b`, die die beiden obigen Randbedingungen nicht verletzen.

Polymorphe Typprüfung und Typinferenz (5)

Veranschaulichung:

```
                          g . f
                        /\     /\
                        /       \
(Int, [b]) -> Int                (a , Char) -> (a , [Char])
      /\                                             /\
       \                                            /
        \                                          /
     Die Eingabe von g        ...ist die Ausgabe von f
```

Somit können wir mittels Unifikation für den allgemeinsten Typ des Kompositionsausdrucks `g . f` schließen:

```
(g . f) :: (Int, Char) -> Int
```

Unifikation im Fall von Beispiel 4

Zentral für die Typschlussfolgerung im Fall von Beispiel 4 ist

- Unifikation.

Veranschaulichung:

```
                          ------------------------------------------
                          |                                        |
  ------------------------|-----------------                       |
 |                        |                  |     (Int,[Int])     |
 |  (Bool,[Char])         |                  |                     |
 |                        |                  |     (Int,[[c]])     |
 |  (c->c,[Char])         |  (Int,[Char])    |                     |
 |                        |                  |     ...             |
 |  ...                   |                  |                     |
 |                        ------------------------------------------
 |                                           |
 |___________________________________________|

     (a,[Char])                                    (Int,[b])

     Ausgabe von f                                 Eingabe von g
```

Unifikation

Wir sagen:

- Eine Instanz eines Typs entsteht durch Ersetzen einer Typvariablen mit einem (konkreten) Typausdruck.
- Eine gemeinsame Instanz zweier Typausdrücke ist ein Typausdruck, der Instanz beider Typausdrücke ist.

Unifikationsproblem:

- Die Suche nach der allgemeinsten gemeinsamen Typinstanz (most general common (type) instance)

Im vorausgegangenen Beispiel 4:

- Die allgemeinste gemeinsame Typinstanz von `(a,[Char])` und `(Int,[b])` ist der Typ `(Int,[Char])`.

Mehr zu Unifikation (1)

Unifikation führt i.a. nicht zu eindeutigen Typen.

Beispiel:

```
(a,[a])         ([b],c)
      \         /
       \/    \/
      ([b],[[b]])
```

`die allgemeinste gemeinsame Instanz`

Beobachtung:

- Randbedingung `(a,[a])` verlangt: Die zweite Komponente ist eine Liste von Elementen des Typs der ersten Komponente.
- Randbedingung `([b],c)` verlangt: Die erste Komponente ist von einem Listentyp.
- Zusammen impliziert dies: die allgemeinste gemeinsame Typinstanz von `(a,[a])` und `([b],c)` ist `([b],[[b]])`.

Mehr zu Unifikation (2)

Beachte:

- Instanz $\neq$ Unifikator
  `([Bool],[[Bool]])` und `([[c]],[[[c]]])` sind beide
  - Instanzen von `([b],[[b]])`.
  - aber keine Unifikatoren: `([b],[[b]])` ist Instanz weder vom einen noch vom anderen.
- Unifikation kann fehlschlagen: So sind beispielsweise `[Int] -> [Int]` und `a -> [a]` nicht unifizierbar.

Mehr zu Unifikation (3)

Detailliertere Veranschaulichung des Unifikationsfehlschlags:

```
[Int] -> [Int]   a -> [a]
       \          /
        \        /
         \/    \/
          (?,?)
```

Beachte: Unifikation der

- Argumenttypen verlangt, dass a vom Typ [Int] ist.
- Resultattypen verlangt, dass a vom Typ Int ist.
- Beide Randbedingungen sind nicht zugleich erfüllbar; Unifikation schlägt deshalb fehl.

Typüberprüfung von Ausdrücken

Beispiel: Polymorphe Funktionsanwendung

```
             f  e
          /\      /\
          /         \
     s -> t           u
     | /\           /\ \
     |  \           /   \

     |   Unifikation    |
     |   von s und u    |

     \/                \/
     s' -> t'          s'
            /\
             \

         Resultattyp t'
```

Beobachtung:

- s und u können verschieden sein; es reicht, wenn sie unifizierbar sind.

Beispiel: `map` und `ord`

Betrachte:

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unifikation von `a -> b` und `Char -> Int` liefert:

```
map :: (Char -> Int) -> [Char] -> [Int]
```

Damit erhalten wir:

```
map ord :: [Char] -> [Int]
```

Beispiel: `foldr` (1)

Betrachte:

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

Anwendungsbeispiel:

```
foldr (+) 0 [3,5,34] = 42
```

Dies legt nahe für den allgemeinsten Typ:

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

Beispiel: `foldr` (2)

Eine eingehendere Überlegung zeigt:

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

Veranschaulichung:

```
                       b
                      /
                     /   a
                    \/  /
foldr f s []      = s  /      b
                      \/ ------------
foldr f s (x:xs)  = f x (foldr f s xs)
                    ------------------
                              b
```

Typüberprüfung polymorpher Fkt.-Definitionen

Für die Typprüfung polymorpher Funktionsdefinitionen

```
f :: t1 -> t2 -> ... -> tk -> t
f p1 p2 ... pk
  | b1 = e1
  | b2 = e2
  ...
  | bn = en
```

sind drei Eigenschaften zu überprüfen:

1. Jeder Wächter b_i muss vom Typ `Bool` sein.
2. Jeder Ausdruck e_i muss einen Wert von einem Typ s_i haben, der wenigstens so allgemein ist wie der Typ `t`, d.h. `t` muss eine Instanz von `si` sein.
3. Das Muster jedes Parameters p_i muss konsistent mit dem entsprechenden Typ t_i sein.

Typüberprüfung und Typklassen

Betrachte:

```
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

In diesem Beispiel erzwingt die Benutzung von (==):

```
member :: Eq a => [a] -> a -> Bool
```

Typüberprüfung und Überladung

Betrachte die Anwendung der Funktion `member` auf `e`:

```
member e
```

mit

```
e :: Ord b => [[b]]
```

Ohne weitere Kontextinformation liefert Unifikation:

```
member :: [[b]] -> [b] -> Bool
```

Somit erhalten wir:

```
member e :: [b] -> Bool
```

Tatsächlich ist weitere Kontextinformation jedoch zu berücksichtigen!

Kontextanalyse

Analyse und Vereinfachung des Kontexts von

```
(Eq [b] , Ord b)
```

liefert:

- Kontextbedingungen beziehen sich auf Typvariablen
  ```
  instance Eq a => Eq [a] where...
  ```
  Dies liefert `(Eq b, Ord b)`.
- Wiederhole diesen Schritt so lange bis keine Instanz mehr anwendbar ist.
- Vereinfachung des Kontexts mithilfe der durch die Typklasse `class` gegebenen Information liefert:
  ```
  class Eq a => Ord a where...
  ```
- Somit: `Ord b`
- Insgesamt: `member e :: Ord b => [b] -> Bool`

Zusammenfassung: Ein dreistufiger Prozess

Der dreistufige Prozess besteht aus:

- Unifikation
- Analyse (mit Instanzen)
- Simplifikation

Dieser Prozess ist typisch für kontextbewusste Analyse in Haskell.

Kapitel 13.3

Typsysteme und Typinferenz

Typsysteme und 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.

    Schlüsselwörter: Typinferenzalgorithmen, Unifikation

Ein typischer Ausschnitt einer Typgrammatik

...erzeugt die Typsprache:

$$\begin{array}{llll} \tau & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Char} \mid \mathit{Bool} & \text{(Einfacher Typ)} \\ & \mid & \alpha & \text{(Typvariable)} \\ & \mid & \tau \to \tau & \text{(Funktionstyp)} \\ \\ \sigma & ::= & \tau & \text{(Typ)} \\ & \mid & \forall\, \alpha.\ \sigma & \text{(Typbindung)} \end{array}$$

Wir sagen:

- τ ist ein Typ.
- σ ist ein Typschema.

Ein typischer Ausschnitt eines Typsystems (1)

...assoziiert mit jedem (typisierbaren) Ausdruck der Sprache einen Typ der Typsprache, wobei Γ eine sogenannte Typannahme ist:

VAR $$\frac{-}{\Gamma \vdash \mathit{var} : \Gamma(\mathit{var})}$$

CON $$\frac{-}{\Gamma \vdash \mathit{con} : \Gamma(\mathit{con})}$$

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 $$\frac{\Gamma[\mathit{var} \mapsto \tau'] \vdash \mathit{exp} : \tau}{\Gamma \vdash \backslash x\text{->}\mathit{exp} : \tau' \to \tau}$$

...

Ein typischer Ausschnitt eines Typsystems (2)

Typannahmen sind

- partielle Funktionen, die Variablen auf Typschemata abbilden.

Dabei ist $\Gamma[var_1 \mapsto \tau_1, \ldots, var_n \mapsto \tau_n]$ die Funktion, die für jedes var_i den Typ τ_i liefert und Γ sonst.

Schematischer Unifikationsalgorithmus

$$
\begin{array}{rcl}
\mathcal{U}(\alpha,\alpha) & = & [\,] \\
\mathcal{U}(\alpha,\tau) & = & \left\{ \begin{array}{ll} [\tau/\alpha] & \text{if } \alpha \notin \tau \\ \mathit{error} & \text{otherwise} \end{array} \right. \\
\mathcal{U}(\tau,\alpha) & = & \mathcal{U}(\alpha,\tau) \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) & = & \mathcal{U}(U\tau_2, U\tau_4)U \\
 & & \quad \text{where } U = \mathcal{U}(\tau_1,\tau_3) \\
\mathcal{U}(\tau,\tau') & = & \left\{ \begin{array}{ll} [\,] & \text{if } \tau = \tau' \\ \mathit{error} & \text{otherwise} \end{array} \right.
\end{array}
$$

Bemerkungen:

- Die Anwendung der Gleichungen erfolgt sequentiell von oben nach unten.
- U ist allgemeinster Unifikator (i.w. eine Substitution).

Unifikationsbeispiel / Allgemeinste Unifikation

Betrachte: `a -> (Bool,c)` und `Int -> b`

- Unifikator: Substitution
  `[Int/a,Float/c,(Bool,Float)/b]`
- Allgemeinster Unifikator: Substitution
  `[Int/a,(Bool,c)/b]`

Beispielanwendung des Unifikationsalgorithmus

Aufgabe:

Unifikation der Typausdrücke `a -> c` und `b -> Int -> a`.

Lösung:

$$
\begin{aligned}
& \mathcal{U}(\mathtt{a\ ->\ c}, \mathtt{b\ ->\ Int\ ->\ a}) \\
(\text{mit } U = \mathcal{U}(\mathtt{a},\mathtt{b}) = [\mathtt{b}/\mathtt{a}]) \quad = \; & \mathcal{U}(U\mathtt{c}, U(\mathtt{Int\ ->\ a}))U \\
= \; & \mathcal{U}(\mathtt{c},\ \mathtt{Int\ ->\ b})[b/a] \\
= \; & [\mathtt{Int\ ->\ b}/\mathtt{c}][\mathtt{b}/\mathtt{a}] \\
= \; & [\mathtt{Int\ ->\ b}/\mathtt{c}, \mathtt{b}/\mathtt{a}]
\end{aligned}
$$

Insgesamt: `b -> Int -> b`

Essenz des automatischen Typinferenzalgorithmus

...ist die syntax-gerichtete Anwendung der Regeln des Typinferenzsystems.

Der Schlüssel:

- Modifikation des Typinferenzsystems derart, dass stets nur eine Regel anwendbar ist.

Zusammenfassung grundlegender Fakten in Haskell zu Typüberprüfung und Typinferenz

- Haskell ist eine stark getypte Sprache
  - Wohltypisierung kann deshalb zur Übersetzungszeit entschieden werden.
  - Fehler zur Laufzeit aufgrund von Typfehlern sind deshalb ausgeschlossen.
- Typen können, müssen aber nicht vom Programmierer angegeben werden.
- Haskell-Interpretierer und -Übersetzer inferieren die Typen von Ausdrücken und Funktionsdefinitionen (in jedem Fall) automatisch.

Resümee

Unifikation

- ist zentral für polymorphe Typinferenz.

Das Beispiel der Funktion magicType

- illustriert nachhaltig 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?

Die Antwort ist ja. Typspezifikationen stellen u.a.

- eine sinnvolle Kommentierung des Programms dar.
- ermöglichen Interpretierer und Übersetzer aussagekräftigere Fehlermeldungen zu erzeugen.

Gezielte Leseempfehlungen (1)

Zu Typen und Typsystemen, Typinferenz:

- Für funktionale Sprachen im allgemeinen
  - Anthony J. Field, Peter G. Robinson. Functional Programming. Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)
- Haskell-spezifisch
  - Simon Peyton Jones, John Hughes. Report on the Programming Language Haskell 98. `http://www.haskell.org/report/`

Gezielte Leseempfehlungen (2)

- Überblick
  - 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.

Gezielte Leseempfehlungen (3)

- Unifikationsalgorithmus
  - J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1):23-42, 1965.
- Typsysteme und Typinferenz
  - Luca Cardelli. Basic Polymorphic Type Checking. Science of Computer Programming 8:147-172, 1987.

Kapitel 13.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 13 (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 13 (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 13 (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 13 (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)

Kapitel 14

Programmierprinzipien

Programmierprinzipien

- Reflektives Programmieren (Kap. 14.1)
  - Stetes Hinterfragen des eigenen Vorgehens
- Funktionen höherer Ordnung (Kap. 14.2)
  - ermöglichen algorithmisches Vorgehen zu verpacken

    Beispiel: Teile und Herrsche
- Funktionen höherer Ordnung plus lazy Auswertung (Kap. 14.3)
  - ermöglichen neue Modularisierungsprinzipien:

    Generator/Selektor-, Generator/Filter-, Generator/Transformator-Prinzip

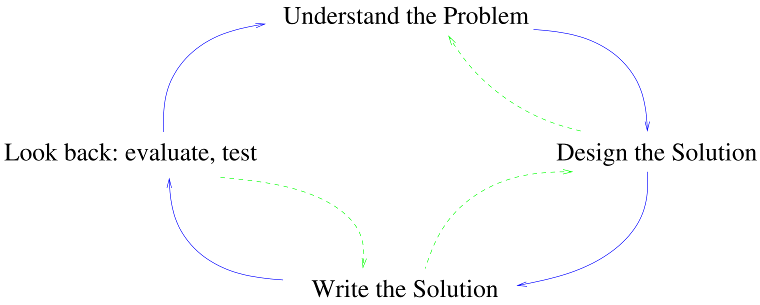
    Beispiel: Programmieren mit Strömen (i.e. unendliche Listen, lazy lists)

Kapitel 14.1

Reflektives Programmieren

Reflektives Programmieren

Der Programm-Entwicklungszyklus nach Simon Thompson, Kap. 11, Reflective Programming, 1999:



- In jeder der 4 Phasen ist es wertvoll und nützlich, (sich) Fragen zu stellen, zu beantworten und ggf. Konsequenzen zu ziehen!

Typische Fragen in Phase 1

Verstehen des Problems:

- Welches sind die Ein- und Ausgaben des Problems?
- Welche Randbedingungen sind einzuhalten?
- Ist das Problem grundsätzlich lösbar?
- Ist das Problem über- oder unterspezifiziert?
- Ist das Problem aufgrund seiner Struktur in Teilprobleme zerlegbar?
- ...

Typische Fragen in Phase 2

Entwerfen einer Lösung:

- Ist das Problem verwandt zu (mir) bekannten anderen, möglicherweise einfacheren Problemen?
- Wenn ja, lassen sich deren Lösungsideen modifizieren und anwenden? Ebenso deren Implementierungen, vorhandene Bibliotheken?
- Lässt sich das Problem verallgemeinern und dann möglicherweise einfacher lösen?
- Ist das Problem mit den vorhandenen Ressourcen, einem gegebenen Budget lösbar?
- Ist die Lösung änderungs-, erweiterungs- und wiederbenutzungsfreundlich?
- ...

Typische Fragen in Phase 3

Ausformulieren und codieren der Lösung:

- Gibt es passende Bibliotheken, insbesondere passende 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?
- ...

Typische Fragen in Phase 4

Blick zurück: Evaluieren, testen:

- Lässt sich die Lösung testen oder ihre Korrektheit sogar 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; würde die Lösung und ihre Implementierung genauso gemacht werden? Was sollte beibehalten oder geändert werden und warum?
- Erfüllt das Programm auch nichtfunktionale Eigenschaften gut wie Performanz, Speicherverbrauch, Skalierbarkeit, Verständlichkeit, Modifizier- und Erweiterbarkeit?
- ...

Kapitel 14.2

Teile und Herrsche

Teile und Herrsche

Gegeben:

Eine Problemspezifikation.

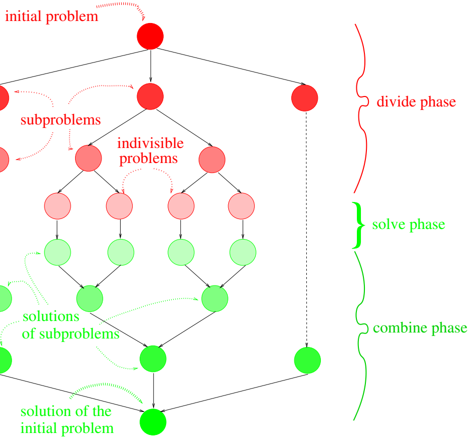
Die (Lösungs-) Idee:

- Ist das Problem elementar (genug), so löse es unmittelbar.
- Anderenfalls zerteile das Probleme in kleinere Teilprobleme und wende diese Zerteilungsstrategie rekursiv an, bis alle Teilprobleme elementar sind.
- Kombiniere die Lösungen der Teilprobleme und berechne darüber die Lösung des ursprünglichen Problems.

- 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.

Die fkt. Umsetzung von Teile und Herrsche (1)

Die Ausgangslage:

- Ein Problem mit
  - Probleminstanzen vom Typ `p`

  und Lösungen mit
  - Lösungsinstanzen vom Typ `s`

Das Ziel:

- Eine Funktion höherer Ordnung `divideAndConquer`, die
  - geeignet parametrisiert Probleminstanzen vom Typ `p` nach dem Prinzip von "Teile und Herrsche" löst.

Die fkt. Umsetzung von Teile und Herrsche (2)

Die Ingredienzien für `divideAndConquer`:

- `indiv :: p -> Bool`: Die Funktion `indiv` liefert `True`, falls die Probleminstanz nicht weiter teilbar ist oder/und nicht mehr weiter geteilt zu werden braucht, weil sie jetzt unmittelbar oder 'hinreichend einfach' lösbar ist.
- `solve :: p -> s`: Die Funktion `solve` liefert die Lösungsinstanz zu einer nicht weiter teilbaren Probleminstanz.
- `divide :: p -> [p]`: Die Funktion `divide` zerteilt eine teilbare Probleminstanz in eine Liste von Teilprobleminstanzen.
- `combine :: p -> [s] -> s`: Angewendet auf eine (Ausgangs-) Probleminstanz und die Liste der Lösungen der zugehörigen Teilprobleminstanzen liefert die Funktion `combine` die Lösung der Ausgangsprobleminstanz.

Die fkt. Umsetzung von Teile und Herrsche (3)

Die Umsetzung:

```
divideAndConquer ::
  (p -> Bool) -> (p -> s) -> (p -> [p]) ->
                            (p -> [s] -> s) -> p -> s

divideAndConquer indiv solve divide combine initPb

 = dAC initPb
   where
    dAC pb
     | indiv pb  = solve pb
     | otherwise = combine pb (map dAC (divide pb))
```

Typische Anwendungen:

- Quicksort, Mergesort
- Binomialkoeffizienten
- ...

Teile und Herrsche am Beispiel von Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort lst
 = divideAndConquer indiv solve divide combine lst
 where
  indiv ls                = length ls <= 1
  solve                   = id
  divide (l:ls)           = [[ x | x <- ls, x <= l],
                              [ x | x <- ls, x > l] ]
  combine (l:_) [l1,l2] = l1 ++ [l] ++ l2
```

Warnung

Nicht jedes Problem, dass sich auf "teile und herrsche" zurückführen lässt, ist auch (unmittelbar) dafür geeignet.

Betrachte:

```
fib :: Integer -> Integer
fib n
 = divideAndConquer indiv solve divide combine n
   where
    indiv n      = (n == 0) || (n == 1)
    solve n
     | n == 0    = 0
     | n == 1    = 1
     | otherwise = error "solve: Problem teilbar"
    divide n     = [n-2,n-1]
    combine _ [l1,l2] = l1 + l2
```

...zeigt exponentielles Laufzeitverhalten!

Ausblick: 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

- Backtracking-Suche
- Prioritätsgesteuerte Suche
- Greedy-Algorithmen
- Dynamische Programmierung

Man spricht hier auch von Algorithmenmustern.

Mehr dazu in der LVA 185.A05 “Fortgeschrittene funktionale Programmierung”.

Kapitel 14.3

Stromprogrammierung

Ströme

Jargon:

- Strom: Synonym für unendliche Liste (engl. lazy list)

Ströme

- erlauben (im Zusammenspiel mit lazy Auswertung) viele Probleme elegant, knapp und effizient zu lösen

Das Sieb des Eratosthenes (1)

Konzeptuell

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 noch nicht gestrichenen Zahl.

Illustration

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

Schritt 2 (Streichen der Vielfachen von "2"):
 2  3     5     7     9     11    13    15    17...

Schritt 2 (Streichen der Vielfachen von "3"):
 2  3     5     7           11    13          17...

Schritt 2 (Streichen der Vielfachen von "5"):...
```

Das Sieb 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,...`

Das Sieb des Eratosthenes (3)

Veranschaulichung: 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,...
->> ...
```

Neue Modularisierungsprinzipien

Aus dem Stromkonzept erwachsen neue Modularisierungsprinzipien.

Insbesondere:

- Generator-/Selektor- (G/S-) Prinzip
- Generator-/Filter- (G/F-) Prinzip
- Generator-/Transformator- (G/T-) Prinzip

Am Beispiel des Siebs des Eratosthenes (1)

Ein Generator (G):

- `primes`

Viele Selektoren (S):

- `take 5`
- `!!42`
- `((take 5) . (drop 5))`
- ...

Am Beispiel des Siebs des Eratosthenes (2)

Zusammenfügen der G/S-Module zum Gesamtprogramm:

- Anwendung des G/S-Prinzips:
  Die ersten 5 Primzahlen:
  ```
  take 5 primes ->> [2,3,5,7,11]
  ```
- Anwendung des G/S-Prinzips:
  Die 42-te Primzahl:
  ```
  primes!!42 ->> 191
  ```
- Anwendung des G/S-Prinzips:
  Die 6-te bis 10-te Primzahl:
  ```
  ((take 5) . (drop 5)) primes
                                  ->> [13,17,19,23,29]
  ```

Am Beispiel des Siebs des Eratosthenes (3)

Ein Generator (G):

- `primes`

Viele Filter (F):

- Alle Primzahlen größer als 1000
- Alle Primzahlen mit mindestens drei Einsen in der Dezimaldarstellung (`hasThreeOnes :: Integer -> Bool`)
- Alle Primzahlen, deren Dezimaldarstellung ein Palindrom ist (`isPalindrom :: Integer -> Bool`)
- ...

Am Beispiel des Siebs des Eratosthenes (4)

Zusammenfügen der G/F-Module zum Gesamtprogramm:

- Anwendung des G/F-Prinzips:
  Alle Primzahlen größer als 1000:
  ```
  filter (>1000) primes
    ->> [1009,1013,1019,1021,1031,1033,1039,...
  ```
- Anwendung des G/F-Prinzips:
  Alle Primzahlen mit mindestens drei Einsen in der Dezimaldarstellung:
  ```
  [ n | n <- primes, hasThreeOnes n]
    ->> [1117,1151,1171,1181,1511,1811,2111,...
  ```
- Anwendung des G/F-Prinzips:
  Alle Primzahlen, deren Dezimaldarstellung ein Palindrom ist:
  ```
  [ n | n <- primes, isPalindrom n]
    ->> [2,3,5,7,11,101,131,151,181,191,313,...
  ```

Am Beispiel des Siebs des Eratosthenes (5)

Ein Generator (G):

- `primes`

Viele Transformatoren (T):

- Der Strom der Quadratprimzahlen
- Der Strom der Primzahlvorgänger
- Der Strom der partiellen Primzahlsummen (d.h. der Strom der Summen der Primzahlen von 2 bis n)
- ...

Am Beispiel des Siebs des Eratosthenes (6)

Zusammenfügen der G/T-Module zum Gesamtprogramm:

- Anwendung des G/T-Prinzips:
  Der Strom der Quadratprimzahlen:

```
[ n*n | n <- primes ]
   ->> [4,9,25,49,121,169,289,361,529,841,...
```

- Anwendung des G/T-Prinzips:
  Der Strom der Primzahlvorgänger:

```
[ n-1 | n <- primes]
   ->> [1,2,4,6,10,12,16,18,22,28,...
```

- Anwendung des G/T-Prinzips:
  Der Strom der partiellen Primzahlsummen:

```
[ sum [2..n] | n <- primes ]
   ->> [2,5,14,27,65,90,152,189,275,434,...
```

Resümee

Lazy Auswertung

- erlaubt es
  - Kontrolle von Daten

  zu trennen und eröffnet dadurch die elegante Behandlung
  - unendlicher Datenwerte (genauer: nicht a priori in der Größe beschränkter Datenwerte), insbesondere
    - unendlicher Listen, sog. Ströme (lazy lists)

Dies führt zu neuen semantisch-basierten, von der Programmlogik her begründeten Modularisierungsprinzipien:

- Generator-/Selektorprinzip
- Generator-/Filterprinzip
- Generator-/Transformatorprinzip

Resümee (fgs.)

Aber Achtung:

- Auf Terminierung ist stets besonders zu achten. So ist

  `filter (<10) primes ->> [2,3,5,7,`

  keine geeignete Anwendung des G/S-Prinzips wegen

  Nichttermination!

  `takeWhile (<10) primes ->> [2,3,5,7]`

  hingegen schon.

Kapitel 14.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (1)

- Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2. Auflage, 1998. (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 14 (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, 2007. (Kapitel 12.6, Modular programming)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 14 (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 14 (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)

Kapitel 15

Fehlerbehandlung

Fehlerbehandlung

...bislang von uns nur ansatzweise behandelt:

- Typische Formulierungen aus den Aufgabenstellungen:

  ...liefert die Funktion diesen Wert als Resultat; anderenfalls:

  - *...endet die Berechnung mit dem Aufruf*
    `error "Ungueltige Eingabe"`
  - *...ist das Ergebnis*
    - *die Zeichenreihe* `"Ungueltige Eingabe"`
    - *die leere Liste* []
    - *der Wert* 0
    - ...
- In diesem Kapitel beschreiben wir Wege zu einem systematischeren Umgang mit unerwarteten Programmsituationen und Fehlern

Typische Fehlersituationen

...sind:

- Division durch 0
- Zugriff auf das erste Element einer leeren Liste
- ...

In der Folge:

- Drei Varianten zum Umgang mit solchen Situationen
  - Panikmodus (Kap. 15.1)
  - Blindwerte (engl. dummy values) (Kap. 15.2)
  - Abfangen und behandeln (Kap. 15.3)

Kapitel 15.1

Panikmodus

Panikmodus (1)

Ziel:

- Fehler und Fehlerursache melden, Berechnung stoppen

Hilfsmittel:

- Die polymorphe Funktion `error :: String -> a`

Wirkung:

- Der Aufruf von

  `error "Fkt f meldet: Ungueltige Eingabe"`

  in Funktion `f` liefert die Meldung

  `Program error: Fkt f meldet: Ungueltige Eingabe`

  und die Programmauswertung stoppt.

Panikmodus (2)

Beispiel:

```
fac :: Integer -> Integer
fac n
 | n == 0    = 1
 | n > 0     = n * fac (n-1)
 | otherwise = error "Fkt fac: Ungueltige Eingabe"

fac 5 ->> 120
fac 0 ->> 1
fac (-5)
 ->> Program error: Fkt fac: Ungueltige Eingabe
```

Panikmodus (3)

Vor- und Nachteile des Panikmodus:

- Schnell und einfach
- Aber: Die Berechnung stoppt unwiderruflich. Jegliche (auch) sinnvolle Information über den Programmlauf ist verloren.

Kapitel 15.2

Blindwerte

Blindwerte (1)

Ziel:

- Panikmodus vermeiden; Programmlauf nicht zur Gänze abbrechen, sondern Berechnung fortführen

Hilfsmittel:

- Verwendung von Blindwerten (engl. dummy values) im Fehlerfall.

Beispiel:

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

Blindwerte (2)

Im Beispiel der Funktion `fac` gilt:

- Negative Werte treten nie als reguläres Resultat einer Berechnung auf.
- Der Blindwert -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 Blindwert in Frage; die Rückmeldung würde so die ungültige Eingabe selbst beinhalten und in diesem Sinn aussagekräftiger und informativer sein.

In jedem Fall gilt:

- Die Fehlersituation ist für den Programmierer transparent.

Blindwerte (3)

Vor- und Nachteile der Blindwertvariante:

- Panikmodus vermieden; Programmlauf wird nicht abgebrochen
- Aber:
  - Oft gibt es einen zwar naheliegenden und plausiblen Blindwert, der jedoch die Fehlersituation verschleiert und intransparent macht.
  - Oft fehlt ein zweckmäßiger und sinnvoller Blindwert auch gänzlich.

Dazu zwei Beispiele.

Blindwerte (4)

Beispiel:

Im Fall der Funktion `tail`

- liegt die Verwendung der leeren Liste `[]` als Blindwert nahe und ist plausibel.

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

Das Auftreten der Fehlersituation wird aber verschleiert und bleibt für den Programmierer intransparent, da

- die leere Liste `[]` auch als reguläres Resultat einer Berechnung auftreten kann

```
tail [42] ->> [] -- reguläres Resultat: keine
                    Fehlersituation
tail []   ->> [] -- irreguläres Resultat: Fehler-
                    situation
```

Blindwerte (5)

Beispiel:

Im Fall der Funktion `head`

- fehlt ein naheliegender und plausibler Blindwert völlig.

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

Mögliche Abhilfe:

- Erweiterung der Signatur und Mitführen des jeweils gewünschten (Blind-) Werts als Argument.

Blindwerte (6)

Beispiel:

Verwende

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

statt (des nicht plausibel Vervollständigbaren):

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

Panikmodus vermieden, aber:

- Keine transparente Fehlermeldung, da der Blindwert hier ebenfalls reguläres Resultat einer Berechnung sein kann.

```
head 'F' "Fehler" ->> 'F' -- reguläres Ergebnis
head 'F' ""       ->> 'F' -- irreguläres Ergebnis
```

Blindwerte (7)

Generelles Muster:

- Ersetze fehlerbehandlungsfreie Implementierung einer (hier einstellig angenommenen) Funktion `f`:

  ```
  f :: a -> b
  f u = ...
  ```

  durch fehlerbehandelnde Implementierung dieser Funktion:

  ```
  f :: b -> a -> b
  f x u
    | errorCondition = x
    | otherwise      = f u
  ```

  wobei `errorCondition` die Fehlersituation charakterisiert.

Blindwerte (8)

Vor- und Nachteile der verfeinerten Blindwertvariante:

- Generalität, stets anwendbar
- Aber: Ausbleibender Alarm: Auftreten des Fehlerfalls bleibt möglicherweise unbemerkt, falls `x` auch als reguläres Ergebnis einer Berechnung auftreten kann

  Konsequenz (mit möglicherweise fatalen Folgen):

  - Vortäuschen eines regulären und korrekten Berechnungsablaufs und eines regulären und korrekten Ergebnisses!
  - Typischer Fall des "Sich ein 'x' für ein 'u' vormachen zu lassen!" (mit möglicherweise fatalen Folgen)!

Kapitel 15.3

Abfangen und behandeln

Abfangen und behandeln (1)

Ziel:

- Abfangen und behandeln von Fehlersituationen

Hilfsmittel:

- Dezidierte Fehlertypen und Fehlerwerte statt schlichter Blindwerte

  Zentral:

```
data Maybe a = Just a
             | Nothing
             deriving (Eq, Ord, Read, Show)
```

  ...i.w. der Typ `a` mit dem Zusatzwert `Nothing`.

Abfangen und behandeln (2)

Damit: Ersetze fehlerbehandlungsfreie Implementierung einer (einstellig angenommenen) Funktion `f :: a -> b` durch:

```
f :: a -> Maybe b
f u
  | errorCondition = Nothing
  | otherwise      = Just (f u)
```

Beispiel:

```
div :: Int -> Int -> Maybe Int
div n m
  | (m == 0)  = Nothing
  | otherwise = Just (n 'div' m)
```

Abfangen und behandeln (3)

Vor- und Nachteile dieser Variante:

- Geänderte Funktionalität: Statt `b`, jetzt `Maybe b`
- Dafür folgender Gewinn:
  - Fehlerursachen können durch einen Funktionsaufruf hindurchgereicht werden:

    $\leadsto$ Effekt der Funktion `mapMaybe`
  - Fehler können gefangen werden:

    $\leadsto$ Aufgabe der Funktion `maybe`

Abfangen und behandeln (4)

- Die Funktion `mapMaybe`:

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just u) = Just (f u)
```

- Die Funktion `maybe`:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f Nothing  = x
maybe x f (Just u) = f u
```

Abfangen und behandeln (5)

Beispiel:

- Fehlerfall: Der Fehler wird hindurchgereicht und gefangen

```
maybe 9999 (+1) (mapMaybe (*3) div 9 0))
->> maybe 9999 (+1) (mapMaybe (*3) Nothing)
->> maybe 9999 (+1) Nothing
->> 9999
```

- Kein Fehler: Alles läuft "normal" ab

```
maybe 9999 (+15) (mapMaybe (*3) div 9 1))
->> maybe 9999 (+15) (mapMaybe (*3) (Just 9))
->> maybe 9999 (+15) (Just 27)
->> 27 + 15
->> 42
```

Abfangen und behandeln (6)

Vor- und Nachteile dieser Variante:

- Fehler und Fehlerursachen können durch Funktionsaufrufe hindurchgereicht und gefangen werden
- Der Preis dafür:
  - Geänderte Funktionalität: `Maybe b` statt `b`

Zusätzlicher pragmatischer Vorteil dieser Variante:

- Systementwicklung ist ohne explizite Fehlerbehandlung möglich.
- Fehlerbehandlung kann am Ende mithilfe der Funktionen `mapMaybe` und `maybe` ergänzt werden.

Kapitel 15.4

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 15

- 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 16

Ein- und Ausgabe

Ein- und Ausgabe

Die Behandlung von

- Ein-/ Ausgabe in Haskell

...bringt uns an die Schnittstelle

- von funktionaler und imperativer Programmierung!

Kapitel 16.1

Einführung und Motivation

Hello, World!

```
helloWorld :: IO ()
helloWorld = putStr "Hello, World!"
```

Hello, World:

- Gewöhnlich eines der ersten Beispielprogramme in einer neuen Programmiersprache
- In dieser LVA erst zum Schluss!

Ungewöhnlich?

Zum Vergleich:

Ein-/Ausgabe-Behandlung in

- Simon Thompson, 1999: In Kapitel 18 (von 20)
- Simon Thompson, 2011: In Kapitel 18 (von 21)
- Peter Pepper, 2003: In Kapitel 21&22 (von 23)
- Richard Bird, 1998: In Kapitel 10 (von 12)
- Antonie J. T. Davie, 1992: In Kapitel 7 (von 11)
- Manuel T. Chakravarty, Gabriele C. Keller, 2004: In Kapitel 7 (von 13)
- ...

Zufall?

...oder ist Ein-/Ausgabe

- weniger wichtig in funktionaler Programmierung?
- in besonderer Weise herausfordernder?

Letzteres:

- Ein-/Ausgabe führt uns an die Schnittstelle von funktionaler und imperativer Programmierung!

Rückblick

Unsere bisherige Sicht funktionaler Programmierung:

Peter Pepper. *Funktionale Programmierung*.
Springer–Verlag, 2003, S.245

In anderen Worten:

Unsere bisherige Sicht funktionaler Programmierung ist

- stapelverarbeitungsfokussiert

...nicht dialog- und interaktionsorientiert wie es gängigen Anforderungen und heutiger Programmierrealität entspricht.

Erinnerung

Im Vergleich zu nicht-deklarativen Paradigmen betont das funktionale Paradigma das

- "was" (Ergebnisse)

zugunsten des

- "wie" (Art der Berechnung der Ergebnisse)

Darin liegt eine wesentliche Stärke deklarativer und speziell auch funktionaler Programmierung!

Erinnerung (fgs.)

Von zentraler Bedeutung in diesem Zusammenhang:

- Kompositionalität

  Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab

  Stichwort: Referentielle Transparenz

  - erleichtert Programmentwicklung und Korrektheitsüberlegungen

- Auswertungsreihenfolgenunabhängigkeit

  Lediglich Auswertungsabhängigkeiten, nicht aber Auswertungsreihenfolgen sind dezidiert festgelegt

  Stichwort: Church-Rosser-Eigenschaft

  - erleichtert Implementierung einschl. Parallelisierung

- ...

Angenommen

...wir erweiterten Haskell naiv um Konstrukte der Art
(*Achtung: Kein Haskell!*):

```
PRINT :: String -> a -> a
PRINT message value =
   << gib "message" am Bildschirm aus und liefere >>
      value

READFL :: Float
READFL = << lies Gleitkommazahl und liefere
            diese als Ergebnis >>
```

Das hieße

…Ein-/Ausgabe in Haskell

- mittels seiteneffektbehafteter Funktionen zu realisieren!

…was den unkontrollierten Verlust von

- Kompositionalität
- Auswertungsreihenfolgenunabhängigkeit

zur Folge hätte.

Verlust der Kompositionalität

Betrachte die Festlegungen von `val`, `valDiff` und `readDiff`

```
val :: Float
val = 3.14

valDiff :: Float
valDiff = val - val

readDiff :: Float
readDiff = READFL - READFL
```

und ihre Anwendung in:

```
constFunOrNot :: Float
constFunOrNot = valDiff + readDiff
```

Verlust der Kompositionalität (fgs.)

Beobachtung:

- Mit der naiven Hinzunahme seiteneffektbehafteter Ein-/Ausgabefunktionen hinge der Wert von Ausdrücken nicht länger ausschließlich von den Werten ihrer Teilausdrücke ab (sondern auch von ihrer Position im Programm)

Somit:

- Verlust der Kompositionalität

  ...und der damit einhergehenden positiven Eigenschaften

Verlust der Auswertungsreihenfolgenunabh.

Vergleiche

```
punkt r = let
            myPi = 3.14
            x    = r * myPi
            y    = r + 17.4
            z    = r * r
            in (x,y,z)
```

...mit (*Achtung: Kein Haskell!*):

```
knackpunkt r = let
                 myPi = PRINT "Constant Value" 3.14
                 u    = PRINT "Erstgelesener Wert" dummy
                 c    = READFL
                 x    = r * c
                 v    = PRINT "Zweitgelesener Wert" dummy
                 d    = READFL
                 y    = r + d
                 z    = r * r
                 in (x,y,z)
```

Verlust der Auswertungsreihenfolgenunabhängigkeit (fgs.)

Beobachtung:

- Mit der naiven Hinzunahme seiteneffektbehafteter Ein-/Ausgabefunktionen hinge der Wert von Ausdrücken nicht länger ausschließlich von den Werten ihrer Teilausdrücke ab (sondern auch von der Reihenfolge ihrer Auswertung)

Somit:

- Verlust der Auswertungsreihenfolgenunabhängigkeit

  ...und der damit einhergehenden Flexibilität

Andererseits

Kommunikation mit dem Benutzer (bzw. der Außenwelt) muss

- die zeitliche Abfolge von Aktivitäten auszudrücken gestatten.

In den Worten von Peter Pepper, 2003:

- "*Der Benutzer lebt in der Zeit und kann nicht anders als zeitabhängig sein Programm beobachten*".

  Das heißt:

  - Man (bzw. ein jedes Paradigma) darf von der Arbeitsweise des Rechners, nicht aber von der des Benutzers abstrahieren!

Nichsdestoweniger

Konzentration auf die Essenz funktionaler Programmierung

- des “was” statt des “wie”

ist wichtig und richtig!

Deshalb darf Realisierung von Ein-/Ausgabe nicht unkontrolliert

- zu Lasten von Kompositionalität und Auswertungsreihenfolgenabhängigkeit

gehen!

Kapitel 16.2

Ein- und Ausgabe in Haskell

Behandlung von Ein-/Ausgabe in fkt. Sprachen

...bringt uns an die Schnittstelle von funktionaler und imperativer Welt!

Zentral für das Vorgehen in Haskell:

- Trennung von
  - rein funktionalem Kern und
  - imperativähnlicher Ein-/Ausgabe

Mittel:

- Elementare Ein-/Ausgabeoperationen (Kommandos) auf speziellen Typen (IO-Typen)
- (Kompositions-) Operatoren, um Anweisungssequenzen (Kommandosequenzen) auszudrücken

Beispiele elementarer Ein-/Ausgabeoperationen

Eingabe:

```
getChar :: IO Char
getLine :: IO String
```

Ausgabe:

```
putChar :: Char -> IO ()
putLine :: String -> IO ()
putStr  :: String -> IO ()
```

Bemerkung:

- `IO`: ...ist Typkonstruktor (ähnlich wie `[·]` für Listen oder `->` für Funktionstypen)
- `IO a`: ...ist spezieller Haskell-Typ "I/O Aktion (Kommando) vom Typ a".
- `()`: ...ist spezieller einelementiger Haskell-Typ, dessen einziges Element (ebenfalls) mit `()` bezeichnet wird.

Einfache Anwendungsbeispiele

Schreiben mit Zeilenvorschub (Standardoperation in Haskell):

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

Lesen einer Zeile und Ausgeben der gelesenen Zeile:

```
echo :: IO ()
echo = getLine >>= putLine -- sog. bind-Operator
                           -- zur Verknüpfung von
                           -- Ein-/Ausgabekommandos
```

Weitere Ein-/Ausgabeoperationen

Schreiben und Lesen von Werten unterschiedlicher Typen:

```
print :: Show a => a -> IO ()
print = putStrLn . show

read :: Read a => String -> a
```

Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion):

```
return :: a -> IO a
```

Erinnerung:

```
show :: Show a => a -> String
```

Kompositionsoperatoren

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>)  :: IO a -> IO b -> IO b
```

Intuitiv:

- (>>=) (oft gelesen als “then” oder “bind”):

  Wenn p und q Kommandos sind, dann ist p >>= q ein zusammengesetztes Kommando, wobei zunächst p ausgeführt wird und einen Rückgabewert x vom Typ a liefert; daran anschließend wird q x ausgeführt, was den Rückgabewert y vom Typ b liefert.

- (>>) (oft gelesen als “sequence”):

  Wenn p und q Kommandos sind, dann ist p >> q das Kommando, das zunächst p ausführt, den Rückgabewert (x vom Typ a) ignoriert, und anschließend q ausführt.

Die do-Notation als (>>=)- und (>>)-Ersatz

...zur bequemeren Bildung von Kommando-Sequenzen:

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"

putTwice :: String -> IO ()
putTwice str = do putStrLn str
                  putStrLn str

putNtimes :: Int -> String -> IO ()
putNtimes n str = if n <= 1
                     then putStrLn str
                     else do putStrLn str
                             putNtimes (n-1) str
```

Weitere Beispiele zur do-Notation

```
putTwice = putNtimes 2

read2lines :: IO ()
read2lines = do getLine
                getLine
                putStrLn "Two lines read."

echo2times :: IO ()
echo2times = do line <- getLine
                putLine line
                putLine line

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

Einmal- vs. Immerwieder-Zuweisung (1)

Durch das Konstrukt

```
var <- ...
```

wird stets eine frische Variable eingeführt.

Sprechweise:

Unterstützung des Konzepts der

- Einmal-Zuweisung (engl. single assignment)

statt des aus imperativen Programmiersprachen bekannten Konzepts der

- Immerwieder-Zuweisung (engl. updatable assignment), der sog. destruktiven Zuweisung

Einmal- vs. Immerwieder-Zuweisung (2)

Zur Illustration des Effekts von Einmal-Zuweisungen betrachte:

```
goUntilEmpty :: IO ()
goUntilEmpty = do line <- getLine
                  while (return (line /= []))
                        (do putStrLn line
                            line <- getLine
                            return () )

while :: IO Bool -> IO () -> IO ()
-- Rumpf von while folgt (siehe Folie Iteration)
```

Lösung:

- ...um trotz Einmal-Zuweisung das intendierte Verhalten ("lies eine Zeile vom Bildschirm ein und gib sie aus, bis die leere Zeile eingelesen wird") zu erhalten:

Rekursion statt Iteration!

Einmal- vs. Immerwieder-Zuweisung (3)

Rekursion:

```
goUntilEmpty :: IO ()
goUntilEmpty =
    do line <- getLine
       if (line == [])
          then return ()
          else (do putStrLn line
                   goUntilEmpty)
```

...von Simon Thompson ([2. Auflage, 1999, S. 393]) vorgeschlagene Lösung mittels Rekursion (statt Iteration).

Einmal- vs. Immerwieder-Zuweisung (4)

Iteration:

```
while :: IO Bool -> IO () -> IO ()

while test action
  = do res <- test
       if res then do action
                      while test action
              else return ()  -- "null I/O-action"
```

Erinnerung:

- Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion):

  `return :: a -> IO a`

Ein-/Ausgabe von und auf Dateien

Auch hierfür gibt es vordefinierte Standardoperatoren:

```
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

wobei

```
type FilePath = String -- implementationsabhängig
```

Anwendungsbeispiel: Bestimmung der Länge einer Datei

```
size :: IO Int
size = do putLine "Dateiname = "
          name <- getLine
          text <- readFile name
          return(length(text))
```

do-Konstrukt vs. (>>=), (>>)-Operatoren

Der Zusammenhang illustriert anhand eines Beispiels:

`incrementInt` mittels do:

```
incrementInt :: IO ()
incrementInt
 = do line <- getLine
      putStrLn (show (1 + read line :: Int))
```

Äquivalent dazu mittels (>>=):

```
incrementInt
 = getLine >>=
   \line -> putStrLn (show (1 + read line :: Int))
```

Intuitiv:

- do entspricht (>>=) plus anonymer λ-Abstraktion

Konvention in Haskell

Einstiegsdefinition (übersetzter) Haskell-Programme

- ist (per Konvention) eine Definition `main` vom Typ `IO a`.

Beispiel:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

Insgesamt:

- `main` ist Startpunkt eines (übersetzten) Haskell-Programms.
- Intuitiv gilt somit:
  "Programm = Ein-/Ausgabekommando"

Resümee über Ein- und Ausgabe

Es gilt:

- Ein-/Ausgabe grundsätzlich unterschiedlich in funktionaler und imperativer Programmierung

  Am augenfälligsten:

  - Imperativ: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich
  - Funktional, speziell in Haskell: Ein-/Ausgabe an bestimmten Programmstellen konzentriert

Häufige Beobachtung:

- Die vermeintliche Einschränkung erweist sich oft als Stärke bei der Programmierung im Großen!

Kapitel 16.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 16 (1)

- 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)
- Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.5, Input/Output in Functional Programming)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 16 (2)

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 16, Communicating with the Outside World)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Kapitel 9, Interactive programs)
- 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)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 16 (3)

- 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)
- 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)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 16 (4)

- 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.

Teil VI

Resümee und Perspektiven

Kapitel 17

Abschluss und Ausblick

Abschluss und Ausblick

Abschluss:

- Rückblick
  - auf die Vorlesung
- Tiefblick
  - in Unterschiede imperativer und funktionaler Programmierung
- Seitenblick
  - über den Gartenzaun auf (ausgewählte) andere funktionale Programmiersprachen

Ausblick:

- Fort- und Weiterführendes

Kapitel 17.1

Abschluss

Abschluss

- Rückblick
  - auf die Vorlesung
- Tiefblick
  - in Unterschiede imperativer und funktionaler Programmierung
- Seitenblick
  - über den Gartenzaun auf (ausgewählte) andere funktionale Programmiersprachen

Rückblick

...unter folgenden Perspektiven:

- Welche Aspekte funktionaler Programmierung haben wir betrachtet?
  - paradigmentypische, sprachunabhängige Aspekte
- Welche haben wir nicht betrachtet oder nur gestreift?
  - sprachabhängige, speziell Haskell-spezifische Aspekte

Vorlesungsinhalte im Überblick (1)

Teil I: Einführung

- Kap. 1: Motivation
  - Ein Beispiel sagt (oft) mehr als 1000 Worte
  - Funktionale Programmierung: Warum? Warum mit Haskell?
  - Nützliche Werkzeuge: Hugs, GHC, Hoogle und Hayoo, Leksah
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 2: Grundlagen von Haskell
  - Elementare Datentypen
  - Tupel und Listen
  - Funktionen
  - Funktionssignaturen, -terme und -stelligkeiten
  - Curry
  - Programmlayout und Abseitsregel
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (2)

- Kap. 3: Rekursion
  - Rekursionstypen
  - Komplexitätsklassen
  - Aufrufgraphen
  - Literaturverzeichnis, Leseempfehlungen

Teil II: Applikative Programmierung

- Kap. 4: Auswertung von Ausdrücken
  - Auswertung von einfachen Ausdrücken
  - Auswertung von funktionalen Ausdrücken
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 5: Programmentwicklung
  - Programmentwicklung
  - Programmverstehen
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (3)

- Kap. 6: Datentypdeklarationen
  - Typsynonyme
  - Neue Typen (eingeschränkter Art)
  - Algebraische Datentypen
  - Zusammenfassung und Anwendungsempfehlung
    - Produkttypen vs. Tupeltypen
    - Typsynonyme vs. Neue Typen
    - Resümee
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (4)

Teil III: Funktionale Programmierung

- Kap. 7: Funktionen höherer Ordnung
  - Einführung und Motivation
  - Funktionale Abstraktion
  - Funktionen als Argument
  - Funktionen als Resultat
  - Funktionale auf Listen
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 8: Polymorphie
  - Polymorphie auf Funktionen
    - Parametrische Polymorphie
    - Ad-hoc Polymorphie
  - Polymorphie auf Datentypen
  - Zusammenfassung und Resümee
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (5)

Teil IV: Fundierung funktionaler Programmierung

- Kap. 9: Auswertungsstrategien

  ...normale vs. applikative Auswertungsordnung, call by name vs. call by value Auswertung, lazy vs. eager Auswertung

  - Einführende Beispiele
  - Applikative und normale Auswertungsordnung
  - Eager oder Lazy Evaluation? Eine Abwägung
  - Eager und Lazy Evaluation in Haskell
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (6)

- Kap. 10: λ-Kalkül
  ...Church-Rosser-Theoreme (Konfluenz, Standardisierung)
  - Hintergrund und Motivation: Berechenbarkeitstheorie und Berechenbarkeitsmodelle
  - Syntax des λ-Kalküls
  - Semantik des λ-Kalküls
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (7)

Teil V: Ergänzungen und weiterführende Konzepte

- Kap. 11: Muster, Komprehensionen und mehr
  - Muster für elementare Datentypen
  - Muster für Tupeltypen
  - Muster für Listen
  - Muster für algebraische Datentypen
  - Das as-Muster
  - Komprehensionen
  - Listenkonstruktoren, Listenoperatoren
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 12: Module
  - Programmieren im Großen
  - Module in Haskell
  - Abstrakte Datentypen
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (8)

- Kap. 13: Typüberprüfung, Typinferenz
  - Monomorphe Typüberprüfung
  - Polymorphe Typüberprüfung
  - Typsysteme und Typinferenz
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 14: Programmierprinzipien
  - Reflektives Programmieren
  - Teile und Herrsche
  - Stromprogrammierung
  - Literaturverzeichnis, Leseempfehlungen
- Kap. 15: Fehlerbehandlung
  - Panikmodus
  - Blindwerte
  - Abfangen und behandeln
  - Literaturverzeichnis, Leseempfehlungen

Vorlesungsinhalte im Überblick (9)

- Kap. 16: Ein- und Ausgabe
  - Einführung und Motivation
  - Ein- und Ausgabe in Haskell
  - Literaturverzeichnis, Leseempfehlungen

Teil VI: Resümee und Perspektiven

- Kap. 17: Abschluss und Ausblick
  - Abschluss
  - Ausblick
  - Literaturverzeichnis, Leseempfehlungen
- Literaturverzeichnis
- Anhänge
  - Formale Rechenmodelle
  - Auswertungsordnungen
  - Datentypdeklarationen in Pascal
  - Hinweise zur schriftlichen Prüfung

Funktionale und imperative Programmierung

Charakteristika im Vergleich:

- Funktional:
  - Programm ist Ein-/Ausgaberelation
  - Programme sind "zeit"-los
  - Programmformulierung auf abstraktem, mathematisch geprägten Niveau
- Imperativ:
  - Programm ist Arbeitsanweisung für eine Maschine
  - Programme sind zustands- und "zeit"-behaftet
  - Programmformulierung konkret mit Blick auf eine Maschine

Funktionale und imperative Programmierung

Charakteristika im Vergleich (fgs.):

- Funktional:
  - Die Auswertungsreihenfolge liegt (abgesehen von Datenabhängigkeiten) nicht fest.
  - Namen werden genau einmal mit einem Wert assoziiert.
  - Neue Werte werden mit neuen Namen durch Schachtelung von (rekursiven) Funktionsaufrufen assoziiert.
- Imperativ:
  - Die Ausführungs- und Auswertungsreihenfolge liegt i.a. fest.
  - Namen können in der zeitlichen Abfolge mit verschiedenen Werten assoziiert werden.
  - Neue Werte können mit Namen durch wiederholte Zuweisung in repetitiven Anweisungen (while, repeat, for,...) assoziiert werden.

Resümee (1)

*"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.

Im Falle von

- Funktionen
  - (Fkt.-) Applikation, Fallunterscheidung und Rekursion
- Datenstrukturen
  - Produkt- und Summenbildung, Rekursion

Resümee (2)

Zusammen mit den Konzepten von

- Funktionen als 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!

Resümee (3)

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

John W. Backus, 1978

Ja!

Im Detail ist zu diskutieren.

Rückblick auf die Vorbesprechung (1)

Was Sie erwarten können:

- 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.

Rückblick auf die Vorbesprechung (2)

Warum es sich für Sie lohnt:

Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
...why the next language you learn should be functional.

Seitenblick

... über den Gartenzaun auf einige ausgewählte andere funktionale Programmiersprachen:

- ML: Ein “eager” Wettbewerber
- Lisp: Der Oldtimer
- APL: Ein Exot

...und einige ihrer Charakteristika.

ML: Eine Sprache mit “eager” Auswertung

ML ist eine strikte funktionale Sprache.

Zu ihren Charakteristika zählt:

- Lexical scoping, curryfizieren (wie Haskell)
- stark typisiert mit Typinferenz, keine Typklassen
- umfangreiches Typkonzept für Module und ADTs
- zahlreiche Erweiterungen (beispielsweise in OCAML) auch für imperative und objektorientierte Programmierung
- sehr gute theoretische Fundierung

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” funktionaler Programmiersprachen

Lisp ist eine noch immer häufig verwendete strikte funktionale Sprache mit imperativen Zusätzen.

Zu ihren Charakteristika zählt:

- einfache, interpretierte Sprache, dynamisch typisiert
- Listen sind gleichzeitig Daten und Funktionsanwendungen
- nur lesbar, wenn Programme gut strukturiert sind
- in vielen Bereichen (insbesondere KI, Expertensysteme) erfolgreich eingesetzt
- 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, angewandt auf die restlichen Listenelemente.

(quote a) bzw. 'a liefert Argument a selbst als Ergebnis.

Beispiele für primitive Funktionen:

(car '(a b c))	->> a	(atom 'a)	->> t
(car 'a)	->> error	(atom '(a))	->> nil
(cdr '(a b c))	->> (b c)	(eq 'a 'a)	->> t
(cdr '(a))	->> nil	(eq 'a 'b)	->> nil
(cons 'a '(b c))	->> (a b c)	(cond ((eq 'x 'y) 'b)	
(cons '(a) '(b))	->> ((a) b)	(t 'c))	->> c

Definition von Funktionen 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: ->> 5
- (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

- kein Curryfizieren in Lisp, 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

Dynamic Scoping vs. Static Scoping

- lexikalisch: Bindung ortsabhängig (Source-Code)
- dynamisch: Bindung vom Zeitpunkt abhängig
- normales Lisp: lexikalisches Binden

  Beispiel:
  ```
  (setq a 100)
  (defun test () a)
  (let ((a 4)) (test))    ⇒ 100
  ```
- dynamisches Binden durch (defvar a) möglich

  obiges 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 vs. Haskell: Ein Vergleich

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	nur eingeschränkt
Typisierung	dynamisch, einfach	statisch, modern
Effizienz	relativ gut	relativ gut
Zukunft	noch lange genutzt	einflussreich

APL: Ein Exot

APL ist eine ältere applikative (funktionale) Sprache mit imperativen Zusätzen.

Zu ihren Charakteristika zählt:

- Dynamische Typisierung
- Verwendung speziellen Zeichensatzes
- Zahlreiche Funktionen (höherer Ordnung) sind vordefiniert; Sprache aber nicht einfach erweiterbar
- Programme sind sehr kurz und kompakt, aber kaum lesbar
- Besonders für Berechnungen mit Feldern gut geeignet

Beispiel: Programmentwicklung in APL

Berechnung der 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. $(2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N)))\ /\ \iota N$

Erfolgreiche Einsatzfelder fkt. Programmierung

- Compiler in compilierter Sprache geschrieben
- Theorembeweiser HOL und Isabelle in ML
- Model-checker (z.B. Edinburgh Concurrency Workbench)
- Mobility Server von Ericson in Erlang
- Konsistenzprüfung mit Pdiff (Lucent 5ESS) in ML
- 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`

Kapitel 17.2

Ausblick

"Alles, was man wissen muss,
um selber weiter zu lernen".

Frei nach (im Sinne von) Dietrich Schwanitz

Fort- und Weiterführendes zu funktionaler Programmierung z.B. in:

- 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. Andreas Frank, Institut für Geoinformation und Kartographie.

LVA 185.A05 Fortg. fkt. Programmierung

Vorlesungsinhalte:

- Monaden und Anwendungen
  - Zustandsbasierte Programmierung
  - Ein-/Ausgabe
  - Parsing
- Kombinatorbibliotheken und Anwendungen
  - Parsing
  - Finanzkontrakte
  - Schöndrucker (Pretty Printer)
- Konstruktorklassen
  - Funktoren
  - Monaden
  - Pfeile
- Funktionale reaktive Programmierung
- Programmverifikation und -validation, Testen
- ...

LVA 185.A05 – Inhaltsübersicht (1)

Part I: Motivation

Part II: Programming Principles

LVA 185.A05 – Inhaltsübersicht (2)

LVA 185.A05 – Inhaltsübersicht (3)

Part III: Quality Assurance

LVA 185.A05 – Inhaltsübersicht (4)

LVA 185.A05 – Inhaltsübersicht (5)

Part IV: Advanced Language Concepts

LVA 185.A05 – Inhaltsübersicht (6)

LVA 185.A05 – Inhaltsübersicht (7)

LVA 185.A05 – Inhaltsübersicht (8)

LVA 185.A05 – Inhaltsübersicht (9)

LVA 185.A05 – Inhaltsübersicht (10)

LVA 127.008 Haskell-Praxis: Programmieren mit der funktionalen Programmiersprache Haskell

Vorlesungsinhalte:

- Analyse und Verbesserung von gegebenem Code
- Weiterentwicklung der Open Source Entwicklungsumgebung für Haskell LEKSAH, insbesondere der graphischen Benutzerschnittstelle (GUI)
- Gestaltung von graphischen 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 17.3

Literaturverzeichnis, Leseempfehlungen

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (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 17 (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, 2007. (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 17 (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)

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Kapitel 17 (4)

- Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.
- 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 Grundlegende, wegweisende Artikel
- III Weitere Artikel
- IV Zum Haskell-Sprachstandard
- V Die Haskell-Geschichte

I 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.
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

Lehrbücher (2)

- 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.
- Chris Done. *Try Haskell*. Online Hands-on Haskell Tutorial. `tryhaskell.org`.

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, 2007.

I Lehrbücher (5)

- Mark P. Jones, Alastair Reid et al. (Hrsg.). *The Hugs98 User Manual*. `www.haskell.org/hugs`
- Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009.
- 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.

I Lehrbücher (6)

- 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.
- 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.

I Lehrbücher (7)

- Peter Rechenberg, Gustav Pomberger (Hrsg.). *Informatik-Handbuch*. Carl Hanser Verlag, 4. Auflage, 2006.
- Colin Runciman, David Wakeling. *Applications of Functional Programming*. UCL Press, 1995.
- 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.

I Lehrbücher (8)

- 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 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.

II 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.

III Weitere Artikel (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.

III Weitere Artikel (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.

III Weitere Artikel (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

III Weitere Artikel (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.

III Weitere Artikel (5)

- 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.
- 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.

III Weitere Artikel (6)

- 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`.
- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.

III Weitere Artikel (7)

- 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.
- John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. Journal of Functional Programming 8(3):275-317, 1998.

III Weitere Artikel (8)

- 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.
- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.

III Weitere Artikel (9)

- 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.
- 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.

III Weitere Artikel (10)

- 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.

III Weitere Artikel (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.

III Weitere Artikel (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.

IV 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`.

V 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

Turing-Maschine

Definition (Turing-Maschine)

- Eine Turing-Maschine 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, die Turing-Maschine 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 (Turing-Tafel)

Eine Turing-Tafel T über einem (endlichen) Zeichenvorrat $\mathcal{A}$ ist eine Tafel mit 4 Spalten und $m + 1$ Zeilen, $m \geq 0$:

$$
\begin{array}{cccc}
i_0 & a_0 & b_0 & j_0 \\
i_1 & a_1 & b_1 & j_1 \\
\ldots & & & \\
i_k & a_k & b_k & j_k \\
\ldots & & & \\
i_m & a_m & b_m & j_m
\end{array}
$$

Turing-Tafel, Turing-Programm (2)

Dabei bezeichnen:

- 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 (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 (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

   $$f_M : Z \to Z$$

   mit

Überführungsfunktion

$\forall\, x \in \mathcal{A}^*,\ \ k \in \mathbb{N}_0$:

$$f_M(x,k) =_{df} \begin{cases} (x, i_k) & \text{falls } k \leq m \text{ und } a_k \text{ keine} \\ & \text{Teilzeichenreihe von } x \text{ ist} \\ (\bar{x} b_k \bar{\bar{x}}, j_k) & \text{falls } k \leq m \text{ und } x = \bar{x} a_k \bar{\bar{x}}, \text{ wobei} \\ & \text{die Länge von } \bar{x} \text{ minimal ist} \\ \textit{undefiniert} & \text{falls } k > m \end{cases}$$

A.3

Primitiv rekursive Funktionen

Primitiv rekursive Funktionen

Definition (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 (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 (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 (Primitive Rekursion)

Ist $f : \mathbb{N}^n \to \mathbb{N}_\perp$ und $g : \mathbb{N}^{n+2} \to \mathbb{N}_\perp$, dann ist $h : \mathbb{N}^{n+1} \to \mathbb{N}_\perp$ 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 (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 (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.
- 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.
- Martin Davis. *What is a Computation?* Chapter in L.A. Steeb (Hrsg.), Mathematics Today – Twelve Informal Essays. Springer-V., 1978.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (3)

- 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.
- Martin Davis. *The Myth of Hypercomputation*. Christof Teuscher (Hrsg.), Alan Turing: Life and Legacy of a Great Thinker, Springer-V., 195-212, 2004.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (4)

- 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.
- 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.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (5)

- 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.
- 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.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (6)

- 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.
- 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.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (7)

- Matjaž Gams. *Alan Turing, Turing Machines and Stronger*. Informatica 37(1):9-14, 2013.
- 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.
- Michael Prasse, Peter Rittgen. *Bemerkungen zu Peter Wegners Ausführungen über Interaktion und Berechenbarkeit*. Informatik-Spektrum 21(3):141-146, 1998.

Vertiefende und weiterführende Leseempfehlungen zum Selbststudium für Anhang A (8)

- 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 (9)

- 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 (10)

- 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 (11)

- 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 (12)

- 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 (13)

- 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

Auswertungsordnungen

B.1

Applikative vs. normale Auswertungsordnung

Applikative vs. normale Auswertungsordnung

Geringe Änderungen können beträchtliche Effekte haben:

- Statt des in Kapitel 9 betrachteten Ausdrucks

  `square (square (square (1+1)))`

  betrachten wir hier den geringfügig einfacheren Ausdruck

  `square (square (square 2))`

  und stellen für diesen Ausdruck die Auswertungsfolgen in
  - applikativer
  - normaler

  Ordnung einander gegenüber.

- Wir werden sehen:
  - Die Zahl der Rechenschritte in (naiver) normaler Auswertungsordnung sinkt erheblich (von 21 auf 14!).

Ausw. in applikativer Auswertungsordnung

...leftmost-innermost (LI) evaluation:

```
              square (square (square 2))
  (LI-E) ->> square (square (2*2))
  (LI-S) ->> square (square 4)
  (LI-E) ->> square (4*4)
  (LI-S) ->> square 16
  (LI-E) ->> 16*16
  (LI-S) ->> 256
```

Insgesamt: 6 Schritte.

Bemerkung:

- (LI-E): Leftmost-Innermost Expansion
- (LI-S): Leftmost-Innermost Simplifikation

Ausw. in normaler Auswertungsordnung

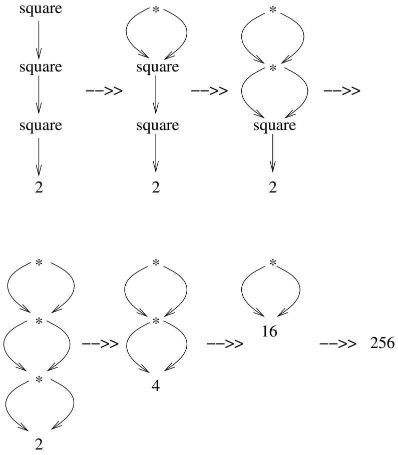
...leftmost-outermost (LO) evaluation:

```
              square (square (square 2))
(LO-E) ->> square (square 2) * square (square 2)
(LO-E) ->> ((square 2)*(square 2)) * square (square 2)
(LO-E) ->> ((2*2)*square 2) * square (square 2)
(LO-S) ->> (4*square 2) * square (square 2)
(LO-E) ->> (4*(2*2)) * square (square 2)
(LO-S) ->> (4*4) * square (square 2)
(LO-S) ->> 16 * square (square 2)
           ->> ...
(LO-S) ->> 16 * 16
(LO-S) ->> 256
```

Insgesamt: 1+6+6+1=14 Schritte.

- (LO-E): Leftmost-Outermost Expansion
- (LO-S): Leftmost-Outermost Simplifikation

Termrepräsentation und -transformation auf Graphen



Insgesamt: 6 Schritte.

(runter von 14 Schritten für (naive) normale Auswertung)

C

Datentypdeklarationen in Pascal

Aufzählungstypen in Pascal

Aufzählungstypen

```
TYPE jahreszeiten = (fruehling, sommer,
                     herbst, winter);
     spielkartenfarben = (kreuz, pik, herz, karo);
     wochenende = (samstag, sonntag);
     geofigur = (kreis, rechteck, quadrat, dreieck);
     medien = (buch, e-buch, dvd, cd);
```

Produkttypen in Pascal

Produkttypen

```
TYPE person = RECORD
                vorname: ARRAY [1..42] OF char;
                nachname: ARRAY [1..42] OF char;
                geschlecht: (maennlich, weiblich);
                alter: integer
              END;

     anschrift = RECORD
                   strasse: ARRAY [1..42] OF char;
                   stadt: ARRAY [1..42] OF char;
                   plz: INT
                   land: ARRAY [1..42] OF char;
                 END;
```

Summentypen in Pascal (1)

Summentypen

```
TYPE multimedia =
 RECORD
  CASE
   medium: medien OF
   buch: (autor, titel: ARRAY [1..100] OF char;
          lieferbar: Boolean);
   e-buch: (autor, titel: ARRAY [1..100] OF char;
            lizenzBis: integer);
   dvd: (titel, regisseur: ARRAY [1..100] OF char;
         spieldauer: real; untertitel: Boolean);
   cd: (interpret, titel, komponist:
        ARRAY [1..100] OF char)
 END;
```

Summentypen in (2)

Summentypen (fgs.):

```
TYPE geometrischefigur =
 RECORD
  CASE
   figur: geofigur OF
   kreis: (radius: real);
   rechteck: (breite, hoehe: real);
   quadrat: (seitenlaenge, diagonale: real);
   dreieck: (seite1, seite2, seite3: real;
             rechtwinklig: Boolean);
 END;
```

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 12.01.2017,** von 16:00 Uhr s.t. bis ca. 18:00 Uhr, im Hörsaal EI7, Gußhausstr. 25-29; die Dauer beträgt 90 Minuten.
- Hilfsmittel: **Keine.**

Hinweise zur schriftlichen LVA-Prüfung (2)

- Anmeldung:
  - Ist erforderlich!
    - Wann: Vorauss. von Mo, 28.11.2016 bis vorauss. Di, 10.01.2017, jeweils 12:00 Uhr
    - Wie: Elektronisch über TISS
- Mitzubringen sind:
  - **Studierendenausweis und Stift**, kein Papier.
- Voraussetzung:
  - Mindestens 50% der Punkte aus dem Übungsteil.

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 2017. Zeugnisausstellung stets zum frühestmöglichen Zeitpunkt; insbesondere nach jedem Klausurantritt; spätestens nach Ablauf des letzten Prüfungstermins.
- Auch zur Teilnahme an der schriftlichen LVA-Prüfung an einem der Nebentermine ist eine Anmeldung über TISS zwingend erforderlich.
- Die genauen Termine werden über TISS angekündigt!