Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2020/2021

Vortrag IV
Orientierung, Einordnung
04.11.2020

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Vortrag IV

Orientierung, Einordnung

...zum selbstgeleiteten, eigenständigen Weiterlernen.

Teil IV: Funktionale Programmierung

- Kapitel 10: Funktionen höherer Ordnung
  $\rightsquigarrow$ Funktionen mit Funktionen als Argument und Resultat.
- Kapitel 11: Polymorphie
  $\rightsquigarrow$ auf Funktionen, Datentypen; echt, unecht, direkt, indirekt.

Plakativ gesprochen

Applikatives Programmieren ist das

- Programmieren und Rechnen mit Funktionen über
  - elementaren Werten.
- Argument und Resultat von Funktionen sind
  - elementare Werte (Zahlen, Zeichen, Wahrheitswerte,...)!

Funktionales Programmieren ist das

- Programmieren und Rechnen mit Funktionen über
  - funktionalen Werten (Funktionen).
- Argument und Resultat von Funktionen sind
  - funktionale Werte (Funktionen) (*sin*, *cos*, *tan*, !, *fib*, $\binom{n}{\cdot}$, $\binom{\cdot}{k}$ $\binom{\cdot}{\cdot}$, `length`, `quickSort`, `curry`,...)!

Teil IV

Funktionale Programmierung

Das Rechnen mit Funktionen und mit Polymorphie

Die zwei (!) Säulen fkt. Programmierung

und

der (!) Hauptleitsatz fkt. Programmierung

Hauptleitsatz funktionaler Programmierung

Maximale Liberalität!

Schränke die Werte von

1. Datentypen
2. Argumenten, Resultaten von Funktionen

und damit auch den Typ von

3. Funktionen

nie mehr ein als für fehlerfreie Auswertung erforderlich!

Was heißt das?

...für die Berechnung

1. der Länge einer Liste, die Anzahl ihrer Elemente,

   ```
   [x,y,z,...]
   ```

   spielt der Typ dieser Elemente keine Rolle; dieser Typ muss daher gar nicht eingeschränkt werden.
2. der Summe zweier Werte

   ```
   x+y
   ```

   reicht es sicherzustellen, dass ihre Werte numerisch sind; eine weitere Einschränkung ihres Typs ist nicht nötig.
3. des Nachfolgers eines Werts

   ```
   succ x
   ```

   reicht es sicherzustellen, dass jeder Wert des Argumenttyps einen eindeutig bestimmten Nachfolger besitzt; eine stärkere Einschränkung des Typs ist nicht nötig.

1. Für das Listenbeispiel heißt das:

Statt vieler auf je einen Elementtyp beschränkter Funktionen:

```
length_il :: [Int] -> Int
length_il []     = 0
length_il (n:ns) = 1 + length_il ns

length_cl :: [Char] -> Int
length_cl (c:cs) = 1 + length_cl cs

length_bl :: [Bool] -> Int
length_bl []     = 0
length_bl (b:bs) = 1 + length_bl bs

length_...l :: [...] -> Int
```

schreiben wir in fkt. Sprachen eine einzige echt polymorphe Fkt.:

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

die auf Listen jedes Elementtyps angewendet werden kann.

2. Für das Summenbeispiel heißt das:

Statt vieler auf je einen numerischen Typ beschränkter Funktionen:

```
plus_i :: Int -> Int -> Int
plus_i m n = m + n
plus_f :: Float -> Float -> Float
plus_f x y = x + y
pus_... :: ... -> ... -> ...
```

gibt es in Haskell die unecht polymorphe Funktion (+) aus der vordefinierten Typklasse Num, der Menge der Typen, deren Werte addiert werden können:

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

Damit sofort möglich:

```
(+) 2 3         ->> 5
(+) 2.35 7.11 ->> 9.46
```

...und – orthogonal dazu – dank zusätzlichen syntaktischen Zuckers zur Anpassung an übliche Schreibgepflogenheiten auch:

```
2 + 3 ->> 5      2.35 + 7.11 ->> 9.46
```

3. Für das Nachfolgerbeispiel heißt das:

Statt vieler auf je einen Typ beschränkter Funktionen, deren Werte aufzählbar sind:

```
succ_i :: Int -> Int
succ_i n = n + 1
succ_c :: Char -> Char
succ_c 'a' = 'b'
succ_c..
succ_...
```

gibt es in Haskell die unecht polymorphe Funktion `succ` aus der vordefinierten Typklasse `Enum`, der Menge der Typen, deren Werte aufgezählt (enumeriert) werden können:

```
succ :: Enum a => a -> a
```

...wird für die Typvariable `a` ein konkreter Typ `T` eingesetzt, der Element (d.h. Instanz) von `Enum` ist, ist `succ` bei der Instanzbildung für `T` so implementiert worden, dass `succ` angewendet auf einen `T`-Wert dessen Nachfolgerwert liefert.

Beachte: Unechte Polymorphie wie von (+)

...kennen wir schon: Unter der Bezeichnung Überladung ist unechte Polymorphie auch in nichtfunktionalen Sprachen üblich.

Allerdings: In Haskell können wir die Überladung von Operatoren bzw. Funktionen einfach u. flexibel auf neue Typen erweitern, z.B.:

```
data Tree = Leaf Int
          | Node Tree Tree deriving (Eq,Show)

instance Num Tree where
 (+) t1 t2 = Node t1 t2
 ...
```

Jetzt können wir auch `Tree`-Werte mit `(+)` 'addieren':

```
t1 = Node (Node (Leaf 2) (Leaf 3)) (Leaf 5)
t2 = Node (Leaf 9) (Leaf 11)
t1 + t2 ->> Node
             (Node (Node (Leaf 2) (Leaf 3)) (Leaf 5))
             (Node (Leaf 9) (Leaf 11))
```

Zurück z. Hauptleitsatz fkt. Programmierung

...wenn Werte möglicher Argumente von Funktionen so wenig wie möglich eingeschränkt werden sollen, müssen auch Funktionen wie:

```
length_funl :: [(Float -> Float)] -> Int
length_funl []     = 0
length_funl (f:fs) = 1 + length_funl fs
```

möglich sein. Und tatsächlich: `length_funl` ist miterfasst von:

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

```
length [sin,cos,tan] ->> 3
```

wie überhaupt jeder Typ als Listenelementtyp, z.B.:

```
length [(+),(-),(*)] ->> 3
length [Leaf 2,Node (Leaf 3) Leaf 5),Leaf 42] ->> 3
length ["Funktionale","Programmierung"] ->> 2
...
```

Statt versteckt "`length [(+),(-),(*)]`"

...können Funktionen auch unmittelbar Argument und auch Resultat von Funktionen sein:

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

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

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

...Funktionen wie `flip`, `curry`, `uncurry`, die Funktionen als Argument oder/und Resultat haben,

- rechnen mit Funktionen!

Solche Funktionen heißen Funktionen höherer Ordnung.

- Funktionen höherer Ordnung rechnen mit Funktionen!

Fkt. höherer Ordnung rechnen mit Funktionen

Funktionen als Argument und Resultat einer Funktion:

```
change :: Num a => (a -> a) -> a -> a -> (a -> a)
change f x y = g where g z = if z == x then y else f z
```

Damit sind Funktionsabänderungen an einer Stelle möglich, z.B.:

```
change fac 0 42 ->> !' ≙ fac'
```

wobei: (math. interpretiert)

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

$$n!' = \begin{cases} 42 & \text{falls } n = 0 \\ n! & \text{falls } n \neq 0 \end{cases}$$

(prog.techn. interpretiert)

```
fac' :: Int -> Int
fac' n
  | n == 0 = 42
  | n /= 0 = fac n
```

Wir können die Funktion (`change fac 0 42`) sofort benutzen:

```
(change fac 0 42) 0 ->> 0!' ≙ fac' 0 ->> 42
(change fac 0 42) 5 ->> 2!' ≙ fac' 5 ->> fac 5 ->> 120
(change fib 0 42) 0 ->> fib' 0 ->> 42
(change fib 0 42) 5 ->> fib' 5 ->> fib 5 ->>  5
```

Bsp. wirkt unmotiviert? Künstlich? Sinnlos?

Dann betrachte:

```
type Nat1             = Int
type Artikelnummer    = Int
type PreisInEURcent   = Nat1
type Warenkatalog     = (Artikelnummer -> PreisInEURcent)
type NeuerPreis       = PreisInEURcent

preisaenderung :: Warenkatalog -> Artikelnummer
                    -> NeuerPreis -> Warenkatalog
preisaenderung warenkatalog artikel neuerPreis
 = change warenkatalog artikel neuerPreis
```

...überzeugt das Beispiel in dieser syntaktischen Verkleidung als sinnvolle Anwendung des Rechnens mit Funktionen?

Nota bene: Die Einschränkung in change

...auf numerische Datentypen ist unnötig restriktiv. Tatsächlich muss nur der Gleichheitstest klappen!

Deshalb können wir den Typ von `change` durch Abschwächen der Kontexteinschränkung sofort verallgemeinern:

```
change :: Eq a => (a -> a) -> a -> a -> (a -> a)
change f x y = g where g z = if z == x then y else f z
```

...weiters kann die Beschränkung auf Funktionen mit identen Argument- und Bildtypen sogar gänzlich fallen:

```
change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g z = if z == x then y else f z
```

...syntaktisch muss auch `g` nicht explizit eingeführt werden:

```
change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = \z -> if z == x then y else f z
```

Polymorphie auf Funktionen und Datentypen

...gibt es in funktionalen Sprachen in gleicher Weise:

Z.B.: Statt vieler auf je einen Blatttyp beschränkter Baumtypen:

```
data Tree_i = Leaf_i Int
            | Node_i Tree_i Tree_i
data Tree_c = Leaf_c Char
            | Node_c Tree_c Tree_c
data Tree_b = Leaf_b Bool
            | Node_b Tree_b Tree_b
...
```

definieren wir in fkt. Sprachen nur einen polymorphen Datentyp für diese strukturidenten Baumtypvarianten:

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

der nicht nur diese vielen, sondern überhaupt alle dieser strukturidenten Baumtypen (Binärbäume mit genau einer Blattbennung) umfasst.

Im Überblick (1)

...das Rechnen mit Funktionen und Polymorphie auf Funktionen und Datentypen sorgen in ihrem nahtlosen Zusammenspiel für viele Einsparungen!

- Eine polymorphe Typdeklaration:

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

statt vieler typspezifischer für jeden benötigten Blatttyp.

- Eine echt, eine unecht polymorphe Funktionsdeklaration:

```
depth :: (Tree a) -> Int
depth (Leaf a)      = 1
depth (Node t1 t2) = 1 + max (depth t1) (depth t2)
```

```
add :: Num a => (Tree a) -> a
add (Leaf a)      = a
add (Node t1 t2) = add t1 + add t2
```

statt vieler typspezifischer für jeden benöt. (numer.) Blatttyp.

Im Überblick (2)

- Eine echt, eine unecht polymorphe Funktion höherer Ordnung:

```
map_tree :: (a -> b) -> (Tree a) -> (Tree b)
map_tree f (Leaf x)      = Leaf (f x)
map_tree f (Node t1 t2) = Node (f t1) (f t2)

filter_tree :: Ord a => (a -> a -> Bool) -> a
                                  -> (Tree a) -> [a]
filter_tree p y (Leaf x)
 | (p y x) && (y < x) = [x]
 | True               = []
filter_tree p y (Node t1 t2)
                      = (filter_tree p y t1)
                        ++ (filter_tree p y t2)
```

statt vieler typspezifischer für jedes benötigte Paar aus Blatt- und Blatttypmanipulationsfunktion bzw. Blatttypprädikat.

Zwei Lesarten für Fkt. höherer Ordnung

...am Beispiel von `map_tree`:

1. Curryfiziert: `map_tree` ist eine Funktion, die eine Funktion von a- in b-Werte auf eine Funktion von `(Tree a)`- in `(Tree b)`-Werte abbildet:

```
map_tree :: (a -> b) -> ((Tree a) -> (Tree b))
map f = g where g (Leaf x)      = Leaf (f x)
                g (Node t1 t2) = Node (f t1) (f t2)
```

2. Uncurryfiziert: `map_tree` ist eine Funktion, die eine Funktion von a- in b-Werte und einen `(Tree a)`-Wert auf einen `(Tree b)`-Wert abbildet:

```
map_tree :: (a -> b) -> (Tree a) -> (Tree b)
map_tree f (Leaf x)      = Leaf (f x)
map_tree f (Node t1 t2) = Node (f t1) (f t2)
```

Kapitel 10

Funktionen höherer Ordnung: Das Rechnen mit Funktionen

Kapitel 10.1

Funktionen als Argument und Resultat von Funktionen: Motivation

Rechnen mit Fkt., die 1. Säule fkt. Programm.

Rechnen mit Funktionen heißt:

Funktionen (*sin*, *cos*, !, *fib*, Warenkataloge,...) können in gleicher Weise wie elementare Werte (Zahlen, Zeichen, Wahrheitswerte, Bäume,...)

- Argument(e)
- Resultat(e)

von Funktionen sein.

Funktionen mit fkt. Argumenten oder/und Resultaten heißen

- Funktionen höherer Ordnung (oder kurz: Funktionale)

Funktionen höherer Ordnung rechnen mit Funktionen, was die

- 1. Säule funktionaler Programmierung

ausmacht!

Ein Beispiel: Das vordefinierte Funktional map

...zur Transformation von Listen in zwei Implementierungsvarianten:

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

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

Funktionen als Argument für map? Kein Problem:

```
map (^2) [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]
map depth [Leaf 2,Node (Leaf 3) Leaf 5),Leaf 42]
            ->> [1,2,1]
```

Das Beispiel fortgesetzt

Funktionen als Argument und Resultat von `map`? Auch kein Problem!

```
map (^) [2,4..10]
 ->> [(2^),(4^),(6^),(8^),(10^)] :: [Int -> Int]
map (*) [2,4..10]
 ->> [(2*),(4*),(6*),(8*),(10*)] :: [Int -> Int]
map (>) [2,4..10]
 ->> [(2>),(4>),(6>),(8>),(10>)] :: [Int -> Bool]
map addiere [Leaf 2,Node (Leaf 3) (Leaf 5),Leaf 42]
 ->> [addiere (Leaf 2),
       addiere (Node (Leaf 3) (Leaf 5)),
       addiere (Leaf 42)] :: [(Tree Int) -> Int]
 ->> [addiere 2,addiere 8,addiere 42]

where addiere :: Num a => (Tree a) -> (Tree a) -> a
      addiere t1 t2 = add t1 + add t2
```

Wir können mit den Funktionen

...der Resultatlisten sofort rechnen:

```
head (map (^) [2,4..10]) 10 ->> (2^) 10  ->> 1024
last (map (*) [2,4..10]) 10 ->> (10*) 10 ->> 100
head (tail (map (^) [2,4..10])) 10 ->> (4>) 10 ->> False
```

```
(map addiere [Leaf 2,Node (Leaf 3) (Leaf 5),Leaf 42])!!1) (Leaf 91)
 ->> addiere (Node (Leaf 3) (Leaf 5)) (Leaf 91)
 ->> (add (Node (Leaf 3) (Leaf 5))) + (add (Leaf 91))
 ->> 8 + 91 ->> 99
```

Eine Handvoll mehr Beispiele

...zum Rechnen mit den Funktionen der Resultatlisten:

```
[f 3 | f <- map (^) [2,4..10]]
  ->> [f 3 | f <- [(2^),(4^),(6^),(8^),(10^)]]
  ->> [2^3,4^3,6^3,8^3,10^3]
  ->> [8,64,216,512,1000]

[f 10 | f <- map (*) [2,4..10]]
  ->> [f 3 | f <- [(2*),(4*),(6*),(8*),(10*)]]
  ->> [2*10,4*10,6*10,8*10,10*10]
  ->> [20,40,60,80,100]

[f 5 | f <- map (>) [2,4..10]]
  ->> [f 3 | f <- [(2>),(4>),(6>),(8>),(10>)]]
  ->> [2>5,4>5,6>5,8>5,10>5]
  ->> [False,False,True,True,True]
```

Zwei Lesarten von map

...als Funktion höherer Ordnung zusammen mit verdeutlichenden Implementierungen:

1. Curryfiziert: `map` ist eine Funktion höherer Ordung, die eine Funktion von a-Werten in b-Werte auf eine Funktion von Listen von a-Werten in Listen von b-Werten abbildet:

```
map :: (a -> b) -> ([a] -> [b])
map f = g where
 g as | as == []       = bs where bs = []
      | as == (a:aas) = bs where bs = ((f a) : g aas)
```

2. Uncurryfiziert: `map` ist eine Funktion höherer Ordung, die eine Funktion von a-Werten in b-Werte und eine Liste von a-Werten auf eine Liste von b-Werten abbildet:

```
map :: (a -> b) -> [a] -> [b]
map f as
 | as == []       = bs where bs = []
 | as == (a:aas) = bs where bs = ((f a) : map f aas)
```

Statt der ‘überdeutlich’ die

...curryfizierte bzw. uncurryfizierte Lesart herausstreichenden Implementierungen:

```
map :: (a -> b) -> ([a] -> [b])      (curryfizierte Lesart)
map f = g where
 g as | as == []       = bs where bs = []
      | as == (a:aas) = bs where bs = ((f a) : g aas)

map :: (a -> b) -> [a] -> [b]       (uncurryfizierte Lesart)
map f as
 | as == []       = bs where bs = []
 | as == (a:aas) = bs where bs = ((f a) : map f aas)
```

...können wir bedeutungsgleich kürzer schreiben:

```
map :: (a -> b) -> ([a] -> [b])      (curryfizierte Lesart)
map f = g where g []     = []
                g (a:as) = (f a) : g as

map :: (a -> b) -> [a] -> [b]       (uncurryfizierte Lesart)
map f []     = []
map f (a:as) = ((f a) : map f as)
```

und die Curryfizierung betonende Variante

...von map:

```
map :: (a -> b) -> ([a] -> [b])      (curryfizierte Lesart)
map f = g where g []     = []
                g (a:as) = (f a) : g as
```

auch mit einer anonymen λ-Abstraktion schreiben können:

```
map :: (a -> b) -> ([a] -> [b])      (curryfizierte Lesart)
map f = g where g = \as -> if as == []
                           then []
                           else (f (head as)) : g (tail as)
```

und dabei noch vorteilhaft vom als-Muster Gebrauch machen:

```
map :: (a -> b) -> ([a] -> [b])      (curryfizierte Lesart)
map f = g where g = \aas@(a:as) -> if aas == []
                                   then []
                                   else (f a) : g as
```

Zum Schluss noch einmal unser Warenhausbsp.

...zum Rechnen mit Funktionen:

```
type Nat1             = Int
type Artikelnummer    = Int
type PreisInEURcent   = Nat1
type Warenkatalog     = (Artikelnummer -> PreisInEURcent)
type NeuerPreis       = PreisInEURcent

preisaenderung :: Warenkatalog -> Artikelnummer
                               -> NeuerPreis -> Warenkatalog
preisaenderung warenkatalog artikel neuerPreis
 = wk where wk = change warenkatalog artikel neuerPreis

change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g = \z -> if z==x then y else f z
```

...erweitert in Vorber. auf eine Rabattschlacht!

```
type Nat0             = Int
type Aktionsartikel   = [Artikelnummer]
type Aktionsrabatt    = Nat0          -- in Prozent, 0 bis 100
type Normalpreis      = PreisInEURcent
type Aktionspreis     = PreisInEURcent
type Aktionskatalog   = Warenkatalog

aktion :: Warenkatalog -> Aktionsartikel
                       -> Aktionsrabatt -> Aktionskatalog
aktion warenkatalog aktionsartikel aktionsrabatt
 = ak where ak = mapfun warenkatalog aktionsartikelPreisListe
       where aktionsartikelPreisListe
             = [(artikel,aktPreis (warenkatalog artikel)
                                  aktionsrabatt)
                  | artikel <- Aktionsartikel]

aktPreis :: Normalpreis -> Aktionsrabatt -> Aktionspreis
aktPreis normpreis aktrab = div (normpreis * (100 - aktrab)) 100

mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f []         = g where g = f
mapfun f ((x,y):zs) = g where g = mapfun (change f x y) zs
```

Probieren Sie es aus! Auf zur Rabattschlacht!

Katalogpreise vor der Rabattschlacht:

```
wk :: Warenkatalog
wk = \n -> if n >= 0 then n * 1000 else (abs n) * 40
```

Eröffnung der Rabattschlacht: Minus 75%-Aktion!

```
wk_aktion :: Aktionskatalog
wk_aktion = aktion wk [artikel | artikel <- [-5,5]] 75
```

Der Schlachtpreisvergleich macht sicher:

```
[(artikel,normalpreis,aktionspreis,ersparnis) |
   artikel <- [-10,10],
   normalpreis  = wk artikel,
   aktionspreis = wk_aktion artikel,
   ersparnis    = normalpreis - aktionspreis]
 ->> [(-10,400,400,0),...,(-5,200,50,150),...,(0,0,0,0),
      ...,(5,5000,1250,3750),...,(10,10000,10000,0)]
```

Nota bene

...die das funktionale Resultat von `mapfun` 'überdeutlich' hervorhebende Implementierung:

```
mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f []           = g where g = f
mapfun f ((x,y):zs) = g where g = mapfun (change f x y) zs
```

würden wir bedeutungsgleich meist kürzer schreiben:

```
mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f []           = f
mapfun f ((x,y):zs) = mapfun (change f x y) zs
```

Natürlich hätten wir

...preisaenderung und aktion auch ohne Abstützung auf change und mapfun direkt definieren können:

```
preisaenderung :: Warenkatalog -> Artikelnummer
                                  -> NeuerPreis -> Warenkatalog
preisaenderung warenkatalog artikel neuerPreis
= wk where wk = \a -> if a == artikel
                      then neuerPreis
                      else warenkatalog a

aktion :: Warenkatalog -> Aktionsartikel
                       -> Aktionsrabatt -> Aktionskatalog

aktion warenkatalog aktionsartikel aktionsrabatt
 = aktionskatalog
    where aktionskatalog
          = \a -> if in_aktion a aktionsartikel
          then (preisaenderung warenkatalog a
                (aktPreis (warenkatalog a) aktionsrabatt)) a
          else warenkatalog a
```

Der Vollständigkeit halber

...hier noch nachgetragen die Implementierung von `in_aktion`:

```
in_aktion :: Artikelnummer -> Aktionsartikel -> Bool
in_aktion _ []      = False
in_aktion a (a':as)
 | a == a'          = True
 | True             = in_aktion a as
```

Kapitel 10.4

Funktionen als Resultat

Kapitel 10.4.2

Methoden 1 bis 6

Funktionen als Resultat: Methode 1

...explizites Ausprogrammieren (in verschiedenen syntaktischen Varianten möglich):

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

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f = g where g = \x y -> f x y

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

iterate :: Int -> (a -> a) -> (a -> a)
iterate n f = g where g x
                      | n > 0     = f . iterate (m-1) f
                      | otherwise = id

extreme :: Ord a => (a -> a -> Bool) -> (a -> a -> a)
extreme p = \x y -> if p x y then x else y
```

Methode 2 bis Methode 6

...siehe vollständige LVA-Unterlagen; auch für weitere Beispiele, für curryfizierte und uncurryfizierte Lesarten komplexer Funktionssignaturen.

Kapitel 10.6

Applikative vs. funktionale Berechnungsweise: Ein Beispiel

Am Beispiel des Algorithmus von Euklid

...zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen veranschaulichen wir den Unterschied zwischen

- applikativer (d.h. Rechnen mit elementaren Werten)
- funktionaler (d.h. Rechnen mit Funktionen)

Berechnungsweise.

Der Algorithmus von Euklid

...zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen $m, n \in \mathbb{N}_1$ ist wie folgt:

1. Wähle x gleich m und y gleich n.
2. Ziehe wiederholt den kleineren der Werte von x und y vom größeren ab.
3. Höre auf, wenn x und y denselben Wert haben. Dieser Wert ist der größte gemeinsame Teiler von m und n.

Applikativ: Rechnen mit elementaren Werten

```
type Nat1 = Integer

ggt_euklid_app :: Nat1 -> Nat1 -> Nat1
ggt_euklid_app x y
  | x > y  = ggt_euklid_app (x-y) y
  | x < y  = ggt_euklid_app x (y-x)
  | x == y = x
```

ggt_euklid_app: Rechnen mit elementaren Werten! Zwei Aufrufbeispiele zur Illustration:

```
m = 18; n = 12
ggt_euklid_app m n ->> ggt_euklid_app 18 12
                   ->> ggt_euklid_app 6 12 (≙ 18-12 12)
                   ->> ggt_euklid_app 6 6 (≙ 6 12-6) ->> 6

m′ = 20; n′ = 35
ggt_euklid_app m′ n′ ->> ggt_euklid_app 20 35
                     ->> ggt_euklid_app 20 15 (≙ 20 35-20)
                     ->> ggt_euklid_app 5 15 (≙ 20-15 15)
                     ->> ggt_euklid_app 5 10 (≙ 5 15-5)
                     ->> ggt_euklid_app 5 5 (≙ 5 10-5) ->> 5
```

Funktional: Rechnen mit Funktionen (1)

```
type Nat1       = Integer
data Variable   = X | Y deriving (Eq,Show)
type Variablen = Variable
type Zustand    = (Variablen -> Nat1)
type Sigma      = Zustand

ggt_euklid_fkt ::        Sigma          ->        Sigma
               ≙ (Variablen -> Nat1) -> (Variablen -> Nat1)
ggt_euklid_fkt sigma
 | sigma X > sigma Y
    = ggt_euklid_fkt (\z -> if z==X then sigma X - sigma Y
                            else sigma Y)
 | sigma X < sigma Y
    = ggt_euklid_fkt (\z -> if z==X then sigma X
                            else sigma Y - sigma X)
 | sigma X == sigma Y = sigma

ggt :: Nat1 -> Nat1 -> Nat1
ggt m n = (ggt_euklid_fkt (\z -> if z==X then m else n)) X
```

Funktional: Rechnen mit Funktionen (2)

ggt_euklid_fkt: Rechnen mit Funktionen! Zwei Aufrufbeispiele zur Illustation:

```
m = 18
n = 12
ggt m n
 ->> (ggt_euklid_fkt sigma1) X
       where sigma1 X = 18
             sigma1 Y = 12
 ->> (ggt_euklid_fkt sigma2) X
       where sigma2 X = 6  (≙ sigma1 X - sigma1 Y ->> 18-12 ->> 6)
             sigma2 Y = 12 (≙ sigma1 Y ->> 12)
 ->> (ggt_euklid_fkt sigma3) X
      where sigma3 X = 6 (≙ sigma2 X ->> 6)
            sigma3 Y = 6 (≙ sigma2 Y - sigma2 X ->> 12-6 ->> 6)
 ->> sigma3 X
 ->> 6
```

Funktional: Rechnen mit Funktionen (3)

```
m′ = 20
n′ = 35
ggt m′ n′
 ->> (ggt_euklid_fkt sigma1) X
       where sigma1 X = 20
             sigma1 Y = 35
 ->> (ggt_euklid_fkt sigma2) X
       where sigma2 X = 20 (≙ sigma1 X ->> 20)
             sigma2 Y = 15 (≙ sigma1 Y - sigma1 X ->> 35-20 ->> 15)
 ->> (ggt_euklid_fkt sigma3) X
       where sigma3 X = 5  (≙ sigma2 X - sigma2 Y ->> 20-15 ->> 5)
             sigma3 Y = 15 (≙ sigma2 Y ->> 15)
 ->> (ggt_euklid_fkt sigma4) X
       where sigma4 X = 5  (≙ sigma3 X ->> 5)
             sigma4 Y = 10 (≙ sigma3 Y - sigma3 X ->> 15-5 ->> 10)
 ->> (ggt_euklid_fkt sigma5) X
       where sigma5 X = 5  (≙ sigma4 X ->> 5)
             sigma5 Y = 5  (≙ sigma4 Y - sigma4 X ->> 10-5 ->> 5)
 ->> sigma5 X
 ->> 5
```

Zur ‘Applikativität’ von ggt_euklid_app

...formal argumentiert ist auch die Funktion ggt_euklid_app eine Funktion, die eine Funktion als Ergebnis hat und insofern mit ‘Funktionen rechnet’ wie die explizite Klammerung der Typsignatur:

```
ggt_euklid_app :: Nat1 -> (Nat1 -> Nat1)
```

und ein Aufruf wie:

```
ggt_euklid_app 42 :: (Nat1 -> Nat1)
```

zeigen.

Die Einführung des uncurryfizierten applikativen Gegenstücks ggt_euklid_app′ zu ggt_euklid_app zeigt, dass das Argument auf der formalen Ebene bleibt und ggt_euklid_app im Kern ‘applikativ’, nicht ‘echt funktional’ ist:

```
ggt_euklid_app' :: (Nat1,Nat1) -> Nat1
ggt_euklid_app' (m,n) = ggt_euklid_app m n
```

Zur ‘Funktionalität’ von ggt_euklid_fkt

...im Unterschied zu ggt_euklid_app ist ggt_euklid_fkt eine Funktion, die wahrhaft mit ‘Funktionen rechnet’, nämlich Funktionen auf Funktionen abbildet:

```
ggt_euklid_fkt :: (Variablen -> Nat1) -> (Variablen -> Nat1)
```

wie die expandierte Typsignatur von ggt_euklid_fkt deutlich macht.

Kapitel 10.7

Zusammenfassung

Zusammenfassung

...Programmierung mit Funktionen höherer Ordnung

- erlaubt das Rechnen mit Funktionen (Stichwort: Funktionen als Argument oder/und Resultat von Funktionen).
- macht das Wesen funktionaler Programmierung aus, die 1. Säule funktionaler Programmierung.

...unterstützt in besonderer Weise:

- Wiederverwendung von Programmcode.
- Kürzere und meist einfacher zu verstehende Programme.
- Einfachere Herleitung, einfacherer Beweis von Programmeigenschaften (Stichwort: Programmverifikation).
- ...

Kapitel 10.8

Leseempfehlungen

Basisleseempfehlungen für Kapitel 10 (1)

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 6, Funktionen höherer Ordnung)

Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 5, Listen und Funktionen höherer Ordnung)

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 5, Polymorphic and Higher-Order Functions; Kapitel 9, More about Higher-Order Functions)

Basisleseempfehlungen für Kapitel 10 (2)


Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 7, Higher-order functions)


Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 5, Higher-order Functions)


Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 4, Functional Programming)


Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 8, Funktionen höherer Ordnung)

Weiterführende Leseempfehlungen für Kap. 10

Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 2.5, Higher-order functional programming techniques)

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 9.2, Higher-order functions: functions as arguments; Kapitel 10, Functions as values; Kapitel 19.5, Folding revisited)

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 11, Higher-order functions; Kapitel 12, Developing higher-order programs; Kapitel 20.5, Folding revisited)

Kapitel 11

Polymorphie: Auf Funktionen, auf Datentypen

Polymorphie, die 2. Säule fkt. Programmierung

...in Haskell unterscheiden wir:

- A Polymorphie auf Funktionen
  - A.1 Echte Polymorphie
  - A.2 Unechte Polymorphie
    - A.2.1 Direkte unechte Polymorphie
    - A.2.2 Indirekte unechte Polymorphie
- B Polymorphie auf Datentypen

...aufgrund der zahlreichen Beispiele zu polymorphen (echt u. unecht) Funktionen und polymorphen Datentypen in vorigen Kapiteln beschränken wir uns hier darauf, die verschiedenen Begriffe noch einmal zu ordnen und im Überblick zu präsentieren.

Polymorphie auf Funktionen und Datentypen macht die

- 2. Säule funktionaler Programmierung

aus!

Kapitel 11.A

Polymorphie auf Funktionen

Kapitel 11.A.1

Echte Polymorphie

Echte Polymorphie

...am Beispiel vordefinierter echt polymorpher Funktionen:

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

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

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

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (a,b) = f a b

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

Echte Polymorphie

...am Beispiel selbstdefinierter echt polymorpher Funktionen:

```
first :: (a,b,c,d) -> a      second :: (a,b,c,d) -> b
first (x,_,_,_) = x          second (_,y,_,_) = y
third :: (a,b,c,d) -> c      fourth :: (a,b,c,d) -> d
third (_,_,z,_) = z          fourth (_,_,_,u) = z
```

```
mapfun_roundrobin :: [(a -> b)] -> [a] -> [b]
mapfun_roundrobin [] _ = []
mapfun_roundrobin _ [] = []
mapfun_roundrobin (f:fs) (x:xs)
            = (f x) : mapfun_roundrobin (fs ++ [f]) xs
```

...auch über selbstdefinierten Datentypen:

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

```
depth :: (Tree a) -> Int
depth (Leaf a)     = 1
depth (Node t1 t2) = 1 + max (depth t1) (depth t2)
```

Echte Polymorphie auf Funktionen

1. ermöglicht die Wiederverwendung von:
   1.1 Funktionsnamen

       ...gute Namen sind knapp!

   1.2 Funktionsimplementierungen

       ...ein&dieselbe Implementierung arbeitet für jeden Typ!
2. wird synonym bezeichnet als:
   - parametrische Polymorphie
3. ist erkennbar an:
   - keine Typvariable der Funktionssignatur ist typkontext-eingeschränkt!

Kapitel 11.A.2

Unechte Polymorphie

Unechte Polymorphie

...am Beispiel vordefinierter unecht polymorpher Funktionen:

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

(==) :: Eq a => a -> a -> a
(/=) :: Eq a => a -> a -> a

(>)     :: Ord a => a -> a -> Bool
(>=)    :: Ord a => a -> a -> Bool
compare :: Ord a => a -> a -> Ordering

succ :: Enum a => a -> a
pred :: Enum a => a -> a

show :: Show a => a -> String
```

Unechte Polymorphie

...am Bsp. selbstdefinierter unecht polymorpher Funktionen:

```
change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y
 = g where g = \z -> if z==x then y else f z

mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f []             = f
mapfun f ((x,y) : zs) = mapfun (change f x y) zs
```

Unechte Polymorphie

...am Bsp. selbstdefinierter unecht polymorpher Funktionen über selbstdefinierten Datentypen:

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

add :: Num a => (Tree a) -> a
add (Leaf x)      = x
add (Node t1 t2) = add t1 + add t2

gleich :: Eq a => (Tree a) -> (Tree a) -> Bool
gleich (Leaf x) (Leaf y) = x == y
gleich (Node t1 t2) (Node t3 t4)
                  = (gleich t1 t3) && (gleich t2 t4)
gleich _ _ = False

zeige :: Show a => (Tree a) -> String
zeige (Leaf x)      = "Blatt " ++ show x
zeige (Node t1 t2) = "Knoten " ++ "<" ++ zeige t1
                              ++ "," ++ zeige t2 ++ ">"
```

Unechte Polymorphie auf Funktionen

1. ermöglicht die Wiederverwendung von:
   - Funktionsnamen

     ...gute Namen sind knapp!
2. nicht aber von Funktionsimplementierungen!

   ...für jeden Typ ist eine eigene typspezifische Implementierung erforderlich!
3. wird synonym bezeichnet als:
   3.1 *ad hoc* Polymorphie
   3.2 Überladung
4. ist erkennbar an:
   - eine oder mehrere Typvariablen der Funktionssignatur sind typkontexteingeschränkt!

Kapitel 11.A.2.1

Direkt unechte Polymorphie

Unecht polymorphe Funktionen

...wie:

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

(==) :: Eq a => a -> a -> a
(/=) :: Eq a => a -> a -> a

(>)      :: Ord a => a -> a -> Bool
(>=)     :: Ord a => a -> a -> Bool
compare :: Ord a => a -> a -> Ordering

succ :: Enum a => a -> a
pred :: Enum a => a -> a

show :: Show a => a -> String
```

heißen

- direkt unecht polymorph

wenn sie Element einer Typklasse (`Num`, `Eq`,...) sind, in einer Typklasse eingeführt sind.

Kapitel 11.A.2.2

Indirekt unechte Polymorphie

Unecht polymorphe Funktionen

...wie:

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

add :: Num a => (Tree a) -> a
add (Leaf x)      = x
add (Node t1 t2) = add t1 + add t2

gleich :: Eq a => (Tree a) -> (Tree a) -> Bool
gleich (Leaf x) (Leaf y) = x == y
gleich (Node t1 t2) (Node t3 t4)
                          = (gleich t1 t3) && (gleich t2 t4)
gleich _ _ = False

zeige :: Show a => (Tree a) -> String
zeige (Leaf x)      = "Blatt " ++ show x
zeige (Node t1 t2) = "Knoten " ++ "<" ++ zeige t1
                        ++ "," ++ zeige t2 ++ ">"
```

heißen

- indirekt unecht polymorph

wenn sie nicht Element einer Typklasse sind, sich aber auf ein solches Element einer Typklasse abstützen (add, zeige,...).

Kapitel 11.A.3

Ausdehnung von Überladung

Statt indirekt unecht polymorpher Funktionen

...wie:

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

gleich :: Eq a => (Tree a) -> (Tree a) -> Bool
gleich (Leaf x) (Leaf y)           = x == y
gleich (Node t1 t2) (Node t3 t4) = (gleich t1 t3)
                                     && (gleich t2 t4)
gleich _ _                         = False

zeige :: Show a => (Tree a) -> String
zeige (Leaf x)      = "Blatt " ++ show x
zeige (Node t1 t2) = "Knoten " ++ "<" ++ zeige t1
                          ++ "," ++ zeige t2 ++ ">"

addiere :: Num a => (Tree a) -> (Tree a) -> a
addiere t1 t2 = add t1 + add t2
```

für die wir uns selbst Funktionsnamen (gleich, zeige,...) ausdenken müssen, können wir auch gut passende Funktionsnamen aus Typklassen wiederverwenden, indem wir sie...

durch passende Instanzbildung für weitere

...Typen überladen und auf deren Werte anwendbar machen:

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

instance Eq a => Eq (Tree a) where
     (Leaf x) == (Leaf y)      = x == y
 (Node t1 t2) == (Node t3 t4) = (t1 == t3) && (t2 == t4)
            _ == _            = False

instance Show a => Show (Tree a) where
 show (Leaf x)      = "Blatt " ++ show x
 show (Node t1 t2) = "Knoten " ++ "<" ++ show t1
                          ++ "," ++ show t2 ++ ">"

instance Num a => Num (Tree a) where
 t1 + t2 = add t1 + add t2
```

Damit sind `(==)`, `show`, `(+)` auch auf `(Tree a)`-Werte anwendbar:

```
t1 = Node (Node (Leaf 2) (Leaf 3)) (Leaf 5)
t2 = Node (Leaf 9) (Leaf 11)

t1 == t2 ->> False                          (statt: gleich t1 t2)
show t2  ->> "Knoten <Blatt 9,Blatt 11>"    (statt: zeige t2)
t1 + t2  ->> 30                             (statt: addiere t1 t2)
```

Noch einfacher: Automatische Instanzbildung

...möglich für exakt die 6 Typklassen `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`, wenn wir als überladene Bedeutung das 'Offensichtliche' haben wollen:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
                                  deriving (Eq,Show)
```

Damit:

```
t1 = Node (Node (Leaf 2) (Leaf 3)) (Leaf 5)
t2 = Node (Leaf 9) (Leaf 11)
t1 == t2 ->> False
show t2  ->> "Node (Leaf 9) (Leaf 11)"
```

...soll aber `show t2` eine 'nicht offensichtliche' Darstellung liefern wie:

```
"Knoten <Blatt 9,Blatt 11>"
```

ist eine entsprechende explizite Instanzbildung erforderlich!

Kapitel 11.B

Polymorphie auf Datentypen

Polymorphie auf Typsynonymen

...polymorphe Typsynonyme:

```
type Paar a b       = (a,b)
type Tripel a b c = (a,b,c)

type Sequenz a = [a]

type Assoziation a b           = Paar a b
type Assoziationssequenz a b = [Assoziation a b]

type Abbildung a b = (a -> b)

type Baum a         = Tree a
type Binaerbaum a = BinaryTree a
```

mit `(Tree a)` aus Kapitel 11.A.1:

```
data Tree a             = Leaf a | Node (Tree a) (Tree a)
newtype BinaryTree a = BT (Tree a)
```

Polymorphie auf Neuen Typen

...polymorphe Neue Typen:

```
newtype Zweitupel a b   = Z (a,b)
newtype Dreitupel a b c = D (a,b,c)

newtype Schlange a = S [a]

newtype Menge a       = M [a]
newtype Relation a b = R [(a,b)]
newtype Funktion a b = F (a -> b)

newtype Relationsmenge a b = RM [[(a,b)]]
newtype Funktionsmenge a b = FM [(a -> b)]

newtype Relationsmenge' a b = RM' [Relation a b]
newtype Funktionsmenge' a b = FM' [F a b]
```

Polymorphie auf algebraischen Typen

...polymorphe algebraische Typen:

```
data BigTree a b c
 = BigLeaf (a -> b)
   | BigNode [c] (BigTree a b c) (BigTree a b c)

data VeryBigTree a b c d
 = VeryBigLeaf d (a -> b -> (c,d))
   | VeryBigNode (Sequenz (b,d)) [BigTree a b c d)]

data TreeOfTree a b c d
 = Nil
   | Branch d [BigTree a b c] (TreeOfTree a b c d)
                              (TreeOfTree a b c d)
```

...zur Illustration von Möglichkeiten, nicht von Nähe zu alltäglichen Anwendungsproblemen.

Typkontexteinschränkungen für polym. Typen

...sind möglich für Neue Typen und algebraische Typen:

```
newtype (Num n, Num m)
          => NumerischeRelation n m = NR [(n,m)]

data (Ord sortierschluessel, Show info)
       => Kartei sortierschluessel info
            = Karteikarte info
              | Unterkartei sortierschluessel
                  (Kartei sortierschluessel info)
                  (Kartei sortierschluessel info)
```

...nicht aber für Typsynonyme.

Polymorphie

...auf Neuen Typen und algebraischen Datentypen

1. ermöglicht die Wiederverwendung von:
   - 1.1 Datenstrukturnamen (Typ- und Konstruktornamen)

     ...gute Namen sind knapp!
   - 1.2 Konstruktionsweise und strukturellem Aufbau für Datenwerte

     ...ein&dieselbe Konstruktionsweise und Struktur für Werte aller Typen!
   - 1.3 polymorphen Funktionen (Funktionsnamen und -implementierungen) auf diesen Datentypen.
2. ist erkennbar an:
   - Typvariablenargumenten des Datentyps, die
     - typkontexteingeschränkt
     - nicht typkontexteingeschränkt

     sein können.

Polymorphie

...auf Typsynonymen

1. ermöglicht die Wiederverwendung:
   1.1 Polymorpher Neuer Typen
   1.2 Polymorpher algebraischer Datentypen
   1.3 Polymorpher Typsynonyme
   1.4 Polymorpher Funktionen (Funktionsnamen und -implementierungen) auf diesen Datentypen
2. ist erkennbar an:
   - Typvariablenargumenten des Typsynonyms, die alle typkontextuneingeschränkt sein müssen

   unter neuen Namen.

Kapitel 11.6

Zusammenfassung

Echte Polymorphie auf Funktionen

- Funktionen, deren Anwendung nicht typklasseneingeschränkt ist, wie `curry`, `change`, `length`, `depth`:

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

change :: (a -> b) -> a -> b -> (a -> b)
change f x y = g
          where g = \z -> if z==x then y else f z

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

data Tree a = Leaf a | Node (Tree a) (Tree a)

depth :: (Tree a) -> Int
depth (Leaf a)     = 1
depth (Node t1 t2) = 1 + max (depth t1) (depth t2)
```

heißen echt polymorph oder parametrisch polymorph.

Unechte Polymorphie auf Funktionen

- Funktionen, die in Typklassen (gleich ob vor- oder eigen-definiert) eingeführt sind, wie `(==)`, `(/=)`:

```
class Eq a where
  (==),(/=) :: a -> a -> a
```

heißen (direkt) unecht polymorph, (direkt) ad hoc polymorph oder (direkt) überladen.

- Funktionen, die typklasseneingeschränkt sind, ohne selbst in einer Typklasse eingeführt zu sein, wie `add`, `addiere`:

```
add :: Num a => (Tree a) -> a
add (Leaf x) = x
add (Node t1 t2) = add t1 + add t2

addiere :: Num a => (Tree a) -> (Tree a) -> a
addiere t1 t2 = t1 + 2
```

heißen indirekt unecht polymorph, indirekt ad hoc polymorph oder indirekt überladen.

Die 2 Säulen funktionaler Programmierung

1. Rechnen mit Funktionen mittels Fkt. höherer Ordnung
2. Polymorphie auf Funktionen und Datentypen

...sind die (!) 2 Säulen funktionaler Programmierung!

Aus ihrem nahtlosen Zusammenspiel wie es in:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = g where g a b = f (a,b)

change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g = \z -> if z==x then y else f z
```

zum Ausdruck kommt, resultieren

- Stärke und Eleganz des funktionalen Programmierstils!

In den Worten von Aristoteles:

Das Ganze ist mehr als die Summe seiner Teile.

Aristoteles (384 - 322 v.Chr.)
griech. Philosoph

Kapitel 11.7

Leseempfehlungen

Basisleseempfehlungen für Kapitel 11


Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 7, Eigene Typen und Typklassen definieren)


Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 3, Types and classes; Kapitel 8, Declaring types and classes)


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)


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)

Weiterführende Leseempfehlungen für Kap. 11

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)

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)

Fazit

über Funktionales Programmieren

Funktionales Programmieren im strengen Sinn

...ist das

- Programmieren und Rechnen auf dem Niveau von Funktionen als Argument und Resultat.
- Aus Funktionen werden mithilfe von Funktionen höherer Ordnung neue Funktionen gebildet.
- Funktionen werden auf Funktionen appliziert; Applikationen von Funktionen auf elementare Werte gibt es nicht.

Summa summarum:

- Das tragende Prinzip funktionalen Programmierens ist die Bildung von Funktionen aus Funktionen mithilfe v. Funktionen höherer Ordnung und die Applikation von Funktionen auf Funktionen; kurz: Das Rechnen mit Funktionen.

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

Umgekehrtes Klassenzimmer III

...zur Übung, Vertiefung

...nach Eigenstudium von Teil III 'Appl. Programmierung':

- Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

- Funktionale Programmierung in der industriellen Praxis

Zwar weiß ich viel...

doch möcht ich alles wissen.

Wagner, Assistent von Faust
Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Zeit für Ihren Zweifel, Ihre Fragen!

Der Zweifel ist der Beginn der Wissenschaft.
Wer nichts anzweifelt, prüft nichts.
Wer nichts prüft, entdeckt nichts.
Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)
franz. Jesuit, Theologe, Geologe und Paläontologe

Die großen Fortschritte in der Wissenschaft
beruhen oft, vielleicht stets, darauf, dass man
eine zuvor nicht gestellte Frage doch,
und zwar mit Erfolg, stellt.

Carl Friedrich von Weizsäcker (1912-2007)
dt. Physiker und Philosoph

...entdecken Sie den Wagner in sich!

Bonusthema

Funktionale Programmierung in der industriellen Praxis

Ausgewählte Einstiegspunkte

- Functional Programming in the Real World.
  `https://homepages.inf.ed.ac.uk/wadler/realworld/`

  Functional is [here] used in the broad sense that includes both 'pure' programs (no side effects) and 'impure' (some use of side effects). Languages covered include CAML, Clean, Erlang, Haskell, Miranda, Scheme, SML, and others...
- Haskell in industry – HaskellWiki.
  `wiki.haskell.org/Haskell_in_industry`

  Haskell in der Industrie, Firmen- und Anwendungsbeispiele.
- Anwendungen funktionaler Programmierung – Wikipedia.
  `en.wikipedia.org/wiki/Functional_programming`
- CUFP – The Annual Commercial Users of Functional Programming Workshop Series. `cufp.org`

  Hauptanwenderkonferenzreihe f. d. Einsatz v. Haskell in d. Industrie.
- GitHub. `github.com`

  Funktionale Programmierung in quelloffenen Projekten.

Ergänzend

...warum setzen sich Programmierstile/Sprachen durch, oder auch nicht? (Achung: Titel irreführend!)

- Philip Wadler. Why no one uses Functional Languages. ACM SIGPLAN Notices 33(8):23-27, 1998.

Hinweis

...für das Verständnis von Vorlesungsteil IV ist eine über den unmittelbaren Inhalt von Vortrag IV hinausgehende weitergehende und vertiefende Beschäftigung mit dem Stoff nötig; siehe:

- vollständige Lehrveranstaltungsunterlagen

...verfügbar auf der Webseite der Lehrveranstaltung:

`http://www.complang.tuwien.ac.at/knoop/fp185A05_ws2021.html`

Aufgabe bis Mittwoch, 18.11.2020

...selbstständiges Durcharbeiten von Teil IV 'Funktionale Programmierung', Kap. 10 und Kap. 11 und von Leit- und Kontrollfragenteil IV zur Selbsteinschätzung und als Grundlage für die umgekehrte Klassenzimmersitzung am 18.11.2020:

Vortrag, umgek. Klassenz.	Thema Vortrag	Thema umgek. Klassenz.
Di, 06.10.2020, 08:15-09:45	Teil I	n.a. / Vorbesprechung
Di, 13.10.2020, 08:15-09:45	Teil II	Teil I
Di, 27.10.2020, 08:15-09:45	Teil III	Teil II
Mi, 04.11.2020, 08:15-09:45	**Teil IV**	**Teil III**
Mi, 18.11.2020, 08:15-09:45	**Teil V**	**Teil IV**
Mi, 02.12.2020, 08:15-09:45	Teil VI	Teil V
Mi, 16.12.2020, 08:15-09:45	Teil VII	Teil VI