



Funktionale Programmierung

Woche 7 – Funktionen höherer Ordnung

René Thiemann

Philipp Dablander

Joshua Ocker

Michael Schaper

Lilly Schönherr

Adam Pescoller

Letzte Vorlesung

- Definition von Typklassen

```
class (...) => TCName a where
  fName :: ty      -- type ty + description of fName
  ...
  lhs = rhs        -- optional default implementation
  ...
```

- Instantiierung von Typklassen

```
instance (...) => TCName (TConstr a1 .. aN) where
  ... -- implementation of functions
```

- Beispiele

- Typklassen: `Eq a`, `Num a`, `Integral a`, `RealFrac a`, ...
- Instanzen: `Integral Int`, `Eq a => Eq (Maybe a)`, `(Ord a, Ord b) => Ord (a,b)`, ...

- Dokumentation:

<http://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html>

- Wechsel zwischen Operator-Symbol und Funktionsname: `(+)` and ``div``

Funktionen Höherer Ordnung

Funktionen und Werte

- Funktionen nehmen Werte als Eingabe und liefern Werte als Resultat
 - bislang sind Werte Zahlen, Zeichen, Paare, Listen, ...
 - Beispiele
 - `lookup :: Eq a => a -> [(a,b)] -> Maybe b`
 - `elem :: Eq a => a -> [a] -> Bool`
- wesentliche Erweiterung: **Funktionen sind Werte**
- Resultat: **Funktionen höherer Ordnung**
 - Funktionen können andere Funktionen als Eingabe haben, z.B.,
`nTimes :: (a -> a) -> Int -> a -> a`
`-- nTimes f n x = f...(f x)`
 - das Resultat einer Funktion kann eine Funktion sein, z.B.,
`compose :: (b -> c) -> (a -> b) -> (a -> c)`
`-- compose f g is the function that takes an x and results in f(g(x))`
- Beobachtung
 - Funktionen höherer Ordnung können auf natürliche Weise definiert werden, z.B.,
`compose f g x = f (g x)`
 - Funktionen höherer Ordnung sind nützlich, um Code-Duplizierung zu vermeiden

Partielle Anwendung

- Frage: wie konstruiert man Werte, die einen Funktionstyp haben?
- eine Möglichkeit: **partielle Anwendung**
- Typkonstruktor ($\rightarrow$) für Funktionen assoziiert nach rechts, siehe [Vorlesung 4, Folie 10](#)

$a \rightarrow b \rightarrow c \rightarrow d$ ist identisch zu $a \rightarrow (b \rightarrow (c \rightarrow d))$

- Anmerkung: Funktions-Anwendung assoziiert nach links

$f \text{ expr1 expr2 expr3}$ ist identisch zu $((f \text{ expr1}) \text{ expr2}) \text{ expr3}$

- Beispiel mit expliziten Klammern

```
average :: Double -> (Double -> Double)
```

```
(average x) y = (x + y) / 2
```

- partielle Anwendung: `average` wird auf weniger als zwei Argumente angewandt
- Beispiel **Ausdrücke**

- `average :: Double -> (Double -> Double)`

keine Argumente angewandt

- `average 3 :: Double -> Double`

1. Argument verfügbar

- `(average 3) 5 :: Double`

zuerst wird 1. Argument übergeben, dann das 2.

- `average 3 5 :: Double`

gleich wie Fall zuvor

Sektionen (sections), `flip`

- **Sektionen** sind eine spezielle Form der partiellen Anwendung eines Operators &
- `(expr &)` ist gleich zu `(&) expr`
- `(& expr)` ist eine Funktion, die ein `x` nimmt und dann zu `x & expr` auswertet
- `(& expr)` ist gleich zu `flip (&) expr`
 - `flip` ist eine vordefinierte Funktion, die die Argumente einer binären Funktion vertauscht

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

- Ausnahme: `(- expr)` ist nicht `flip (-) expr`, sondern nur der negierte Wert von `expr`
- Beispiele
 - `(> 3)` teste, ob eine Zahl größer als 3 ist
 - `(3 >)` teste, ob 3 größer als eine Zahl ist
 - `(3 -)` ziehe etwas von 3 ab
 - `(- 3)` die Zahl -3

Beispiel: nTimes

```
nTimes :: (a -> a) -> Int -> a -> a
```

```
nTimes f n x
```

```
  | n == 0 = x
```

```
  | otherwise = f (nTimes f (n - 1) x)
```

- Beobachtungen

- `nTimes` benutzt gewöhnliche Rekursion auf Zahlen
- in der letzten Zeile wird `f` zweimal genutzt
 - einmal als Parameter von `nTimes`,
wo in `nTimes f` das `f` selbst nicht auf ein Argument angewandt wird
 - einmal als eine Funktion, die auf ein Argument angewandt wird: `otherwise = f (...)`

- Anwendung: präzise Definition anderer Funktionen

```
tower :: Integer -> Int -> Integer -- tower x n = x ^ (x ^ ... (x ^ 1))
```

```
tower x n = nTimes (x ^) n 1 -- n exponentiations with basis x
```

```
replicate :: Int -> a -> [a] -- replicate n x = [x, ..., x]
```

```
replicate n x = nTimes (x :) n [] -- n insertions of x
```

Auswertung bei Partieller Anwendung

- wenn die linke Seite einer definierende Gleichung von `f` die Gestalt `f pat1 ... patN` mit `N` Argumenten hat, dann kann `f expr1 ... exprM` nur ausgewertet werden, wenn $M \geq N$
- Beispiel `nTimes` und `tower`

```
nTimes f n x
| n == 0 = x
| otherwise = f (nTimes f (n - 1) x)
tower x n = nTimes (x ^) n 1
```

```
tower 4 2
= nTimes (4 ^) 2 1      -- here, (4 ^) cannot be evaluated
= 4 ^ (nTimes (4 ^) 1 1) -- ^-invocation with 2 arguments is created
= 4 ^ (4 ^ (nTimes (4 ^) 0 1)) -- another ^-invocation
= 4 ^ (4 ^ 1)
= 4 ^ 4
= 256
```

Auswertung bei Partieller Anwendung, fortgesetzt

- wenn die linke Seite einer definierende Gleichung von f die Gestalt $f \text{ pat1 } \dots \text{ patN}$ mit N Argumenten hat, dann kann $f \text{ expr1 } \dots \text{ exprM}$ nur ausgewertet werden, wenn $M \geq N$
- Beispiel mit $M > N$

```
selectFunction :: Bool -> (Int -> Int) -- same as Bool -> Int -> Int
selectFunction True  = (* 3)
selectFunction False = abs
```

```
    selectFunction False (-2)  -- M > N
= abs (-2)
= 2
```

- Einschränkung: alle definierenden Gleichung einer Funktion müssen die gleiche Anzahl an Argumenten aufweisen
- Konsequenz: das folgende Programm wird nicht akzeptiert, obwohl es Sinn machen würde

```
selectFunction' :: Bool -> Int -> Int
selectFunction' True  = (* 3)
selectFunction' False x = 2 - x
```

Curry-Form (currying)

- bislang haben wir die meisten Funktionen in **Curry-Form** definiert (Haskell B. Curry, M. Schönfinkel)

$$f :: \text{ty1} \rightarrow \dots \rightarrow \text{tyN} \rightarrow \text{ty}$$

- Alternative ist **Tupel-Form**, die logisch äquivalent ist

$$f :: (\text{ty1}, \dots, \text{tyN}) \rightarrow \text{ty}$$

- Beobachtung

- partielle Anwendung ist nur bei Curry-Form möglich
- Tupel-Form hat Vorteil, wenn zusammengehörige Werte übergeben werden sollen

```
type Date = (Int, Int, Int)
```

```
differenceDate :: Date -> Date -> Int -- number of days between two dates  
-- but not: Int -> Int -> Int -> Int -> Int -> Int -> Int
```

- Reihenfolge der Argumente ist bei Curry-Form wichtig:
partielle Anwendung findet immer von links nach rechts statt

- 1000 durch etwas dividieren:
- Division durch 1000:
- Alternative mit **flip**:

```
div 1000  
let f x = div x 1000 in f  
flip div 1000
```

- Daumenregel: Argumente einer Funktion, die sich selten ändern, sollten weit links stehen

η -Äquivalenz

- betrachten Sie eine Funktion zur Auswertung von mathematischen Ausdrücken

```
type Var = String
data Expr = Variable Var | Number Int | Plus Expr Expr | ...
eval :: [(Var,Int)] -> Expr -> Int
-- defining equations of eval
```

- Aufgabe: definieren Sie eine Funktion `evalxy`, die Ausdrücke für eine fixe Variablen-Belegung auswertet, in der x mit 2 und y mit 7 belegt ist

```
evalFix e = eval [("x",2), ("y",7)] e
```

- Beobachtung: linke und rechte Seite haben das gleiche letzte Argument: `e`
- η -Äquivalenz besagt, dass `evalFix` äquivalent zu `evalFixR` ist

```
evalFixR = eval [("x",2), ("y",7)]
```
- η -Kontraktion: ändere `f x = expr x` zu `f = expr`, falls `x` nicht in `expr` vorkommt
- η -Expansion: ändere `f = expr` zu `f x = expr x`, falls `f` ein Funktionstyp ist
- für Haskell-Programme wird häufig η -Kontraktion genutzt: kürzere Programme

Anonyme Funktionen: λ -Abstraktionen

- Beispiel: wende die Funktion, die zu einem x den Wert $3 \cdot (x + 1)$ berechnet, n mal an
- eine Möglichkeit: lokale Definition einer Funktion

```
example :: Num a => Int -> a -> a
```

```
example n y = let f x = 3 * (x + 1) in nTimes f n y
```

```
-- or even shorter via eta-contraction
```

```
example = let f x = 3 * (x + 1) in nTimes f
```

- störend: Bedarf eines Funktions-Namens, hier f
- Alternative: Erzeugung einer anonymen Funktion mittels λ -Abstraktion

- Syntax: $\backslash \text{pat1} \dots \text{patN} \rightarrow \text{expr}$

λ wird als $\backslash$ in Haskell notiert

- äquivalent zu: $\text{let } f \text{ pat1} \dots \text{patN} = \text{expr} \text{ in } f$

wobei f ein frischer Name ist

```
example = nTimes (\ x -> 3 * (x + 1))
```

- Unterschied zwischen λ -Abstraktionen und lokal definierten Funktionen
 - λ -Abstraktionen können nicht rekursiv sein
 - λ -Abstraktionen benötigen keine neuen Funktionsnamen

Beispiele von Funktionen höherer Ordnung und deren Anwendung

Definition von Abstrakten und Wiederverwendbaren Programmen

- betrachten Sie folgende Aufgaben
 - multiplizieren Sie alle Listen Elemente mit 2
 - ersetzen Sie alle Buchstaben in einem String durch Großbuchstaben
 - berechnen Sie eine Liste von eMail Adressen aus einer Liste von Studierenden
- mögliche Implementierungen

```
multTwo [] = []
```

```
multTwo (x : xs) = 2 * x : multTwo xs
```

```
toUpperList [] = []
```

```
toUpperList (c : cs) = toUpper c : toUpperList cs
```

```
eMails [] = []
```

```
eMails (s : ss) = getEmail s : eMails ss
```

- Beobachtung: alle Funktionen sind sehr ähnlich
- abstrakte Sicht: wende eine Funktion auf jedes Listen-Element an
- Ziel: einmalige Implementierung des allgemeinen Schemas, was dann für jede Einzel-Aufgabe wiederverwendet werden kann

Die `map` Funktion

- `map` wendet eine Funktion auf jedes Listen-Element an

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

```
map f [] = []
```

```
map f (x : xs) = f x : map f xs
```

- bessere Lösung der Aufgaben der vorigen Folie

```
multTwo = map (2 *)
```

```
toUpperList = map toUpper
```

```
eMails = map getEmail
```

- Beispiel-Auswertung ist analog zur Auswertung bei partieller Anwendung

```
toUpperList "Hi"  
= map toUpper "Hi"  
= toUpper 'H' : map toUpper "i"  
= 'H' : toUpper 'i' : map toUpper ""  
= 'H' : 'I' : ""  
= "HI"
```

Die filter Funktion

- `filter` selektiert alle Elemente einer Liste, die eine gewisse Bedingung erfüllen

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

```
filter f [] = []
```

```
filter f (x : xs)
```

```
    | f x =    x : filter f xs
```

```
    | otherwise = filter f xs
```

- Beispiel-Anwendung

```
-- check, if an element occurs in a list
```

```
elem :: Eq a => a -> [a] -> Bool
```

```
elem x xs = filter (== x) xs /= []
```

```
-- another implementation of the lookup-function
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

```
lookup x xs = case filter (\ (k,_) -> x == k) xs of
```

```
    [] -> Nothing
```

```
    ((_,v) : _) -> Just v
```

Anwendung: Quicksort

- quicksort ist ein effizienter Sortier-Algorithmus
- Idee: Teile nicht-leere Liste in kleine und große Elemente, und sortiere dann rekursiv
- kompakte Implementierung

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) =    -- x is pivot element
    qsort (filter (<= x) xs) ++ [x] ++ qsort (filter (> x) xs)
```

- Implementierung kann auf mehrere Arten optimiert werden
 - einmalige Nutzung von `partition :: (a -> Bool) -> [a] -> ([a], [a])`, anstelle von zwei Aufrufen von `filter`
 - flexible Wahl der Ordnung
 - `qsortBy :: (a -> a -> Bool) -> [a] -> [a]`
 - `qsort = qsortBy (<=)`
 - **zufällige** Wahl des Pivot Elements, siehe Vorlesung Algorithmen und Datenstrukturen oder VU Fortgeschrittene Funktionale Programmierung

Der Funktions-Kompositions Operator (.)

- Komposition von Funktionen ist eine Funktion höherer Ordnung (in Haskell: `(.)`)
`(.) :: (b -> c) -> (a -> b) -> (a -> c)`
`(f . g) = \ x -> f (g x)`
- die Komposition nimmt zwei Funktionen als Eingabe und liefert Funktion
- Funktions-Komposition wird oft genutzt, um mehrere Funktionen nacheinander anzuwenden, ohne Zwischen-Ergebnisse in Variablen zu speichern
- Beispiel: gegeben eine Zahl, addiere 5, dann berechne den Absolutwert, dann multipliziere mit 7, konvertiere in einen String, und berechne dessen Länge
- ohne Komposition: viele Klammern, nicht sehr lesbar
`\ x -> length (show ((abs (x + 5)) * 7))`
- klar und präzise mit Funktions-Komposition
`length . show . (* 7) . abs . (+ 5)`

Mengen-Sicht

- Listen bieten Möglichkeit, Mengen (mit Duplikaten) von Elementen zu repräsentieren
- dann kann man die gesamte Menge mittels `map`, `filter`, `sum`, ... bearbeiten, ohne auf die Position der Elemente in der Liste zu achten
- der n -tes-Element-Operator (`!!`) sollte hier **nicht** verwendet werden
- insbesondere sollten Sie das folgende imperative Programm

```
for (int i = 0; i < length; i++) {  
    xs[i] = someFun(xs[i]);  
}
```

nicht als funktionales Programm

```
map (\ i -> someFun (xs !! i)) [0 .. length xs - 1]    -- bad
```

implementieren, sondern einfach als:

```
map someFun xs                                           -- good
```

- das schlechte Programm benötigt $\sim \frac{1}{2}n^2$ Auswertungs-Schritte für eine Liste der Länge n :
Listen $\neq$ Arrays!

Anwendung: Namen guter Studierenden

- gegeben eine Liste von Studierenden, berechnen Sie eine sortierte Liste aller Namen von Studierenden, deren Durchschnitts-Note 2 oder besser ist
- Implementierung

```
data Student = ...  
avgGrade :: Student -> Double  
...  
getName :: Student -> String  
...
```

```
goodStudents :: [Student] -> [String]  
goodStudents = qsort . map getName . filter (\ s -> avgGrade s <= 2)
```

Zusammenfassung

- Funktionen höherer Ordnung
 - Funktionen dürfen Funktionen als Eingabe haben
 - Funktionen dürfen Funktionen als Ausgabe haben
- partielle Anwendung
 - eine n -stellige Funktion ist ein Wert
 - Anwendung einer n -stelligen Funktion auf 1 Argument resultiert in $n - 1$ -stelliger Funktion
 - Sektionen sind eine spezielle Syntax für partiell angewandte Operatoren
- η -Kontraktion: kontrahiere $f \dots x = \text{expr } x$ zu $f \dots = \text{expr}$
- λ -Abstraktionen sind anonyme Funktionen
- bearbeiten Sie Listen, die Mengen repräsentieren, mittels `map`, `filter`, ..., aber nicht mit (!!)