Wiederholung Currying

Idee:

Jede Funktion mit mehreren Argumenten lässt sich als geschachtelte Funktionen mit je einem Argument auffassen (und aufschreiben)

Beispiel:

Die folgenden Zeilen definieren dieselbe Funktion vom Typ

```
g :: Int -> Int -> Bool
```

- `g m n = m < n` $((g\ m)\ n) = m < n$
- `g m = \ n -> m < n` $(g\ m) = \lambda n.(m < n)$
- `g = \ m n -> m < n` $g = \lambda m.\lambda n.(m < n)$

mit Argument-Tupel (Achtung: anderer Typ):

```
g' :: (Int, Int) -> Bool
g' (m, n) = m < n
```

in mathematischer Notation (bekannt aus Modellierung):

zweistellig: $C^{(A \times B)}$ ist isomorph zu $(C^B)^A$

$(n-1)$-stellig: $A_n^{(A_1 \times \cdots \times A_{n-1})}$ ist isomorph zu $\left(\cdots\left(A_n^{A_{n-1}}\right)\cdots\right)^{A_1}$

Wiederholung Funktionen höherer Ordnung

Funktionen als Argument von Funktionen

Beispiel:

```
twice :: (a -> a) -> a -> a
twice = \ f x -> f (f x)
double :: Int -> Int
double = \ x -> 2 * x
```

Anwendung:

`twice double` hat den Typ `Int -> Int`

Herleitung: (allgemeiner) Typ von `twice` ist

```
twice :: (a -> a) -> a -> a
```

Typ von `double` (hier erstes Argument von `twice`) ist

```
double :: Int -> Int
```

Passende Instanziierung von `twice` ist

```
twice :: (Int -> Int) -> Int -> Int
```

Typ von `twice double` ist also (Currying)

```
twice double :: Int -> Int
```

Analog hat `twice double 3` den Typ `Int` und den Wert 12

Wiederholung Funktionen höherer Ordnung

```
twice :: (a -> a) -> a -> a
twice = \ f x -> f (f x)
succ :: Int -> Int
succ =  \ x -> x + 1
```

mehr Beispiele:

- `\x -> 2 * x + 1` hat den Typ `Int -> Int`
- für `twice (\x -> 2 * x + 1)` mit Typ `Int -> Int` hat `twice (\x -> 2 * x + 1) 3` Typ `Int` und Wert 15
- `succ 0, twice succ 0, twice twice succ 0` (ÜA)
- `twice twice succ 0, twice twice twice succ 0` (ÜA)
- `twice (^2) 3, twice twice (^2) 3`
- Typ von `twice twice` und `twice twice twice` ? (ÜA)

Funktionen höherer Ordnung – Anwendungen

z.B. für

- punktweise Summe zweier Funktionen:

```
fsum :: (a -> Int) -> (a -> Int) -> (a -> Int)
fsum f g x = (f x) + (g x)
fsum f g = \x -> (f x) + (g x)
```

Beispiele:

- `fsum (*2) (+1) 4`,
- `fsum len head [ 2 .. 5 ]`

- Komposition von Funktionen (Achtung: Reihenfolge)

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

Beispiele:

- `( ( \ x -> x * 2 ) . length ) "foo"`
- `( ( * 2 ) . length ) "foo"`
- `suchbaum = sortiert . inorder`

Wiederholung: rekursive Datentypen

- Peano-Zahlen

```
data Nat = Z
         | S Nat
```

- Listen

```
data List a = Nil {}
         | Cons { head :: a, tail :: List a}
```

oder kürzer

```
data List a = Nil | Cons a (List a)
```

- Binärbäume

```
data Tree a = Leaf {}
         | Branch { left :: Tree a,
                    key :: a,
                    right :: Tree a}
```

oder kürzer

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

Wiederholung: Funktionen auf rekursiven Datentypen

Entwurf rekursiver Funktionen auf rekursiven Datentypen:

1. Typdefinition
2. Angabe aller Basis- und rekursiven Fälle
3. Definition der Ergebnisse der Basisfälle
4. Definition der Ergebnisse der rekursiven Fälle
5. evtl. Typ verallgemeinern

Beispiel: Summe aller Schlüssel eines Baumes

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

1. Typdefinition: `tsum :: Tree Int -> Int`
2. Angabe aller Basis- und rekursiven Fälle:

```
tsum t = case t of
    Leaf          -> ...
    Branch l k r -> ...
```

3. Definition der Ergebnisse der Basisfälle: `Leaf -> 0`
4. Definition der Ergebnisse der rekursiven Fälle:

```
Branch l k r -> (tsum l) + k + (tsum r)
```

Funktionen auf rekursiven Datentypen – Beispiele

Operationen auf

- Peano-Zahlen:
  - Verdoppeln
  - Test auf gerade
  - Addition, Multiplikation, ...
- Listen:
  - Summe aller Listenelemente (`fold`)
  - Länge der Liste (`fold`)
  - Verkettung (`fold`)
  - Verdoppeln jedes Listenelements (`map`)
  - Liste aller geraden Listenelemente (`filter`)
- Bäumen:
  - Verdoppeln jedes Schlüssels
  - Angabe gerade / ungerade für jeden Schlüssel
  - Anzahl aller Schlüssel
  - Summe aller Schlüssel
  - Inorder-Durchquerung

Wiederholung: Rekursionsschema `fold` für Listen

Beschreibung des Rekursionsmusters für Listen

```
g :: List a -> b
g xs = case xs of
  Nil         -> s
  Cons x xss -> f x ( g xss )
```

mit zwei Funktionen als Argumenten:
`s :: b` und `f :: a -> b -> b`

Funktion höherer Ordnung

```
fold :: ( a -> b -> b ) -> b -> List a -> b
```

Anwendung: `g = fold f s`
ermöglicht kurze Funktionsdefinition, z.B.

```
sum :: List Int -> Int
sum = fold (+) 0
doubles :: List Int -> List Int
doubles = fold (\ x y -> Cons (2*x) y ) Nil
evens :: List Int -> List Int
evens = fold (\ x y -> if (mod x 2 == 0)
                       then Cons x y else y ) Nil
```

Funktionen höherer Ordnung für Listen

Beispiele für in Haskell vordefinierte Funktionen höherer Ordnung zur Verarbeitung von Listen:

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

und viele Spezialfälle davon:

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

WH: Rekursionsmuster über Peano-Zahlen

```
data Nat = Z | S Nat

double :: Nat -> Nat
double x = case x of
    Z    -> Z
    S x' -> S ( S (double x' ) )

even :: Nat -> Bool
even x = case x of
    Z    -> True
    S x' -> not ( even x' )
```

gemeinsames Schema:

```
g :: Nat -> b
g x = case x of
    Z    -> ...
    S x' -> ...
```

Rekursionsschema `fold` über Peano-Zahlen

Rekursionsschema:

```
g :: Nat -> b
g x = case x of
    Z    -> z
    S x' -> s ( g x' )
```

mit zwei Funktionen als Parametern
`z :: b` und `s :: b -> b`
`fold` über Peano-Zahlen:

```
fold :: b -> (b -> b) -> Nat -> b
fold z s x = case x of
    Z    -> z
    S x' -> s ( fold z s x' )
```

Beispiele:

- `double = fold Z ( S . S )`
- `even = fold True not`

WH: Rekursionsmuster über Bäume

```
data Tree a     = Leaf
 | Branch { left :: Tree a, key :: a, right :: Tree a }

doubles :: Tree Int -> Tree Int
doubles t = case t of
    Leaf           -> Leaf
    Branch l k r -> Branch (doubles l) (k*2) (doubles r)

preorder :: Tree a -> [a]
preorder t = case t of
    Leaf           -> []
    Branch l k r -> [ k ]
                    ++ ( preorder l )
                    ++ ( preorder r )

sum :: Tree Int -> Int
sum t = case t of
    Leaf           -> 0
    Branch l k r -> ( sum l ) + k + ( sum r )
```

Rekursionsschema `map` über Bäume

```
f :: Tree a -> Tree b
f t = case t of
    Leaf           -> Leaf
    Branch l k r -> Branch (f l) (g k) (f r)
```

Beispiel:

```
f = doubles
g = double
```

Rekursionsschema:

```
map :: (a -> b ) -> Tree a -> Tree b
map f t = case t of
     Leaf -> Leaf
     Branch l k r -> Branch (map f l)
                            (f k)
                            (map f r)


doubles = map ( 2* )
```

Rekursionsschema `fold` über Bäume

```
f :: Tree a  -> b
f t = case t of
    Leaf           -> leaf
    Branch l k r -> branch (f l) k (f r)
```

Beispiel:

```
f = preorder
leaf = []
branch l' k r' = [k] ++ l' ++ r'
```

Rekursionsschema:

```
fold :: b -> (b -> a -> b -> b) -> Tree a -> b
fold leaf branch t = case t of
    Leaf           -> leaf
    Branch l k r -> branch (fold leaf branch l)
                           k
                           (fold leaf branch r)
```

Beispiele: `fold` über Bäume

```
fold :: b -> (b -> a -> b -> b) -> (Tree a) -> b
fold leaf branch t = case t of
    Leaf          -> leaf
    Branch l k r -> branch (fold leaf branch l)
                           k
                           (fold leaf branch r)
```

```
preorder = fold [] ( \ l' k r' -> [k] ++ l' ++ r' )
sum      = fold 0 ( \ l' k r' -> l' + k + r' )
```

analog: Anzahl der Blätter, inneren Knoten, Tiefe

Rekursionsmuster (Merksätze)

Rekursionsmuster anwenden =
jeden Konstruktor durch eine passende Funktion ersetzen

- Anzahl der Muster-Argumente =
  Anzahl der Konstruktoren
  (plus eins für das Datenargument)
- Stelligkeit eines Muster-Argumentes =
  Stelligkeit des entsprechenden Konstruktors
- Rekursion im Typ $\rightarrow$ Rekursion im Muster
- zu jedem rekursiven Datentyp gibt es genau ein passendes Rekursionsmuster

`fold` über nichtrekursiven Datentypen – Bool

```
data Bool = False | True
g :: Bool -> Int
g x = case x of
  False -> 5
  True  -> 3
```

Muster

```
g :: Bool -> Int
g x = case x of
  False -> f
  True  -> t
```

mit Funktionen `f,t :: Int` (Konstanten)

```
fold :: b -> b -> Bool -> b
fold f t x = case x of
  False -> f
  True  -> t
```

Beispiele:

- `g = fold 5 3`
- `not = fold True False`

`fold` über nichtrekursiven Datentypen – Maybe

```
data Maybe a = Nothing | Just a
g :: Maybe Int -> Bool
g x = case x of
  Nothing -> False
  Just z  -> z > 3
```

Muster

```
g :: Maybe a -> b
g x = case x of
  Nothing -> n
  Just z  -> j z
```

mit Funktionen `n :: b, j :: a -> b`

```
fold :: b -> ( a -> b ) -> Maybe a -> b
fold n j x = case x of
  Nothing -> n
  Just z  -> j z
```

Beispiele:

- `g = fold False ( > 3 )`
- `not = fold True False`