# Programming Paradigms

## Summer Term 2017

7th Lecture

**Prof. Janis Voigtländer**
**University of Duisburg-Essen**

# Programming Paradigms

## Parametric polymorphism

# Parametrically polymorphic functions

- Many of the already seen/existing functions on lists are meant for lists over arbitrary element types, e.g.:

```
length [ ]        =  0
length (x : xs)  =  length xs + 1
```

```
> length [ 1, 2 ]
2
```

```
> length [ [ ], [ 'a', 'b', 'c' ] ]
2
```

- Like for standard functions, one naturally would like to have the same flexibility for one’s own defined functions:

```
concatenation  [ ]          ys   =   ys
concatenation  (x : xs)  ys   =   x : concatenation xs ys
```

# Parametrically polymorphic functions

Instead of several variants:

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{concatenation} :: [\mathrm{Int}] \rightarrow [\mathrm{Int}] \rightarrow [\mathrm{Int}]} \\
\mathrm{concatenation}\ [\,] & \mathrm{ys} & = & \mathrm{ys} \\
\mathrm{concatenation}\ (\mathrm{x} : \mathrm{xs}) & \mathrm{ys} & = & \mathrm{x} : \mathrm{concatenation\ xs\ ys}
\end{array}$$

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{concatenation}' :: [\mathrm{Bool}] \rightarrow [\mathrm{Bool}] \rightarrow [\mathrm{Bool}]} \\
\mathrm{concatenation}'\ [\,] & \mathrm{ys} & = & \mathrm{ys} \\
\mathrm{concatenation}'\ (\mathrm{x} : \mathrm{xs}) & \mathrm{ys} & = & \mathrm{x} : \mathrm{concatenation}'\ \mathrm{xs\ ys}
\end{array}$$

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{concatenation}'' :: \mathrm{String} \rightarrow \mathrm{String} \rightarrow \mathrm{String}} \\
\mathrm{concatenation}''\ [\,] & \mathrm{ys} & = & \mathrm{ys} \\
\mathrm{concatenation}''\ (\mathrm{x} : \mathrm{xs}) & \mathrm{ys} & = & \mathrm{x} : \mathrm{concatenation}''\ \mathrm{xs\ ys}
\end{array}$$

only one definition:

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{concatenation} :: [\mathrm{a}] \rightarrow [\mathrm{a}] \rightarrow [\mathrm{a}]} \\
\mathrm{concatenation}\ [\,] & \mathrm{ys} & = & \mathrm{ys} \\
\mathrm{concatenation}\ (\mathrm{x} : \mathrm{xs}) & \mathrm{ys} & = & \mathrm{x} : \mathrm{concatenation\ xs\ ys}
\end{array}$$

# Type variables and parametrized types

- In order to be able to assign types to polymorphic functions, one uses variables that act as place holders for arbitrary types:

type variables

- With type variables, we can build parametrized types for polymorphic functions:

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

- If the result type is also described via a type variable, then of course the concrete type of the actual parameter determines the type of the result:

```
> :t last
last :: [a] → a
```

```
> :t last [ True, False ]
last [ True, False ] :: Bool
```

# Safe use of polymorphic functions

```
concatenation :: [a] → [a] → [a]
concatenation [ ]          ys  =  ys
concatenation (x : xs)  ys  =  x : concatenation xs ys
```

```
> concatenation [ True ] [ False, True, False ]
[True, False, True, False]

> concatenation "abc" "def"
"abcdef"

> concatenation "abc" [True]
    Couldn't match 'Char' against 'Bool'
        Expected type: Char
        Inferred type: Bool
    In the list element: True
    In the second argument of 'concatenation', namely '[True]'
```

# Further examples

```
drop :: Int → [Int] → [Int]
drop 0         xs        =  xs
drop n         [ ]       =  [ ]
drop (n + 1) (x : xs)  =  drop n xs
```

```
drop :: Int → [a] → [a]
drop 0         xs        =  xs
drop n         [ ]       =  [ ]
drop (n + 1) (x : xs)  =  drop n xs
```

```
zip :: [Int] → [Int] → [(Int, Int)]
zip (x : xs) (y : ys)  =  (x, y) : zip xs ys
zip xs         ys         =  [ ]
```

```
zip :: [a] → [b] → [(a, b)]
zip (x : xs) (y : ys)  =  (x, y) : zip xs ys
zip xs         ys         =  [ ]
```

```
fst :: (a, b) → a

head :: [a] → a

take :: Int → [a] → [a]

id :: a → a
```

# Safe use of polymorphic functions

```
zip :: [a] → [b] → [(a, b)]
zip (x : xs) (y : ys)  =  (x, y) : zip xs ys
zip xs       ys        =  [ ]
```

```
> zip "abc" [ True, False, True ]
[('a', True), ('b', False), ('c', True)]

> :t "abc"
"abc" :: [Char]

> :t [ True, False, True ]
[True, False, True] :: [Bool]

> :t [ ('a', True), ('b', False), ('c', True) ]
[('a', True), ('b', False), ('c', True)] :: [(Char, Bool)]
```

# Polymorphic data types

Abstraction possible from:

```
data Tree = Leaf Int | Node Tree Int Tree
```

to:

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

with data type constructors typed as follows:

```
> :t Leaf
Leaf :: a → Tree a
> :t Node
Node :: Tree a → a → Tree a → Tree a
```

# Polymorphic data types

- Possible values for:

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

are, for example: Leaf 3 :: Tree Int
Node (Leaf 'a') 'b' (Leaf 'c') :: Tree Char

but not: Node (Leaf 'a') 3 (Leaf 'c')

- Example function:

```
height :: Tree a → Int
height (Leaf _)          =  0
height (Node t1 _ t2)  =  1 + max (height t1)  (height t2)
```

# Polymorphic type synonyms

Same kind of abstraction possible for type:

```
type PairList a b = [(a, b)]
```

# Programming Paradigms

## Ad-hoc polymorphism

## Standard type classes, in particular automatic "deriving"

- The generous introduction of ever new types might seem unattractive at first, given that one then also has to (re-)implement certain functionality over and over again (e.g., for input and output, for computing on enumeration types, …).
- But these concerns are dispelled by mechanisms providing generic functionality, for example:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

allColors  =  [minBound .. maxBound] :: [Color]
```

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr  deriving (Read, Show, Eq)
```

…

# Standard type classes

Best explained through examples:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

instance Show Color where
  show Red      =  "rot"
  show Green  =  "gruen"
  …
```

```
data Rat  =  Rat (Int, Int)

instance Show Rat where
  show (Rat (n, m))  =  show n ++ " / " ++ show m
```

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr

instance Show Expr where
  show (Lit n)          =  "Lit "  ++  show n   ++  "; "
  show (Add e1 e2)  =  show e1   ++  show e2   ++  "Add; "
  show (Mul e1 e2)  =  show e1   ++  show e2   ++  "Mul; "
```

Of course, arbitrary other functions can also be called, not only the one currently being defined (on the same or on another type).

# Interplay with parametric polymorphism

- We used type variables to express that a certain functionality does not depend, say, on the type of elements of a list:

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

- How does that now play out for show?

- Certainly we do not want to write something like:

```
instance Show [Int] where
  show [ ]        =  "[ ]"
  show (i : is)  =  … show i … show is …

instance Show [Color] where
  show [ ]         =  "[ ]"
  show (c : cs)  =  … show c … show cs …
```

# Interplay with parametric polymorphism

- Parametrization over the element type, but with constraint on the type variable:

```
instance Show a ⇒ Show [a] where
  show [ ]        =  "[ ]"
  show (x : xs)  =  … show x … show xs …
```

- Such a constraint can also express a dependency on <u>another</u> type class:

```
instance Show a ⇒ Eq a where
  x == y   =   show x == show y
```

- And in a very natural way, constraints can also appear in the type signatures of “normal” functions:

```
elem :: Eq a ⇒ a → [a] → Bool
elem x [ ]         =  False
elem x (y : ys)  =  x == y  | |  elem  x  ys
```

# User defined type classes?

- First, a look at the definitions of two classes in the standard library:

```
class Eq a where
  (==) :: a → a → Bool
  x == y  =  not (x /= y)
  (/=) :: a → a → Bool
  x /= y  =  not (x == y)

class Eq a ⇒  Ord a where
  (<), (<=), (>), (>=) :: a → a → Bool
  …
```

optional default implementations

- Here Eq a ⇒  Ord a does <u>not</u> mean that every Eq-type <u>is</u> also an Ord-type, but instead that a type only <u>can</u> belong to type class Ord if it already belongs to type class Eq. (And, naturally, if it moreover supports operations (<), (<=), …, for whose default implementations one can of course make use of the assumed existing Eq-functionality.)

- Definition of one's own type classes is simply analogous (see live examples).

# Programming Paradigms

## Higher-Order Functions

# Higher-Order: functions as parameters and results (of other functions)

- In Haskell functions may “manipulate” or “generate” other functions:

  - Functions may be function arguments.
  - Functions may be function results.

- Name for this kind of functions (corresponding to concepts from predicate logic):

  functions of higher order

- Functions that only process or generate “normal data” are called functions of first order.

# Currying (1)

- In Haskell, functions with multiple parameters are usually viewed as being implicitly “staged” functions of only one parameter each (saving parentheses):

```
between :: Integer → (Integer → (Integer →  Bool))
between  x  y  z      | x <= y && y <= z  =  True
                      | otherwise         =  False
```

- Application of this principle is now called currying (after Haskell B. Curry, who studied this technique extensively, though the original “inventor” is actually the logician Schönfinkel).

- The above form of the between-function is called the “curried” form, while the more conventional (mathematics style) form with a parameter tuple is called “uncurried”.

# Currying (2)

- Beside saving parentheses, the curried notation has the advantage that of each function, one automatically has available several variants (with different arities).

```
between :: Integer → (Integer → (Integer →  Bool))
between  x  y  z        | x <= y && y <= z   =   True
                        | otherwise          =   False
```

```
between  2        ::  Integer → (Integer → Bool)
between  2  3     ::  Integer → Bool
between  2  3  4  ::  Bool
```

- Each such partial application has all “rights” of a function, in particular may itself be further applied, passed on, stored in a data structure, …

# Partial applications of operators: “sections”

- An operator that normally is written between its arguments can be turned into a (curried) function to be written in front of its arguments, simply by enclosing it in parentheses:

```
> (+) 3 4
7
```

- Called a “section”, it is also possible to include one of the arguments directly:

```
> (/) 3 2
1.5
```

vs.

```
> (3/) 2
1.5
```

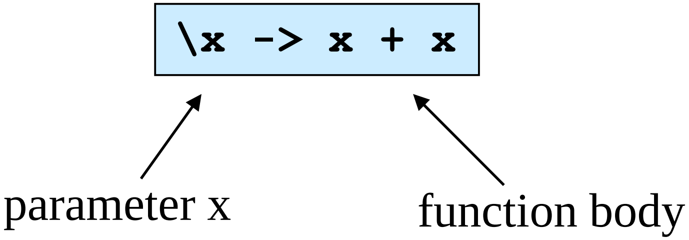
vs.

```
> (/2) 3
1.5
```

- Some further examples: (>3), (1+), (1/), (*2), (++ [42])

# Anonymous functions (1)

- Functions can be created anonymously, that is, without giving them a name. For example:



```
\x -> x + x
```

- This corresponds to the mathematical notation of "λ-abstractions", e.g.:

```
λx.(x + x)
```

- Their application is treated like normal function evaluation, e.g.:

```
> (\x → x + x) 3
6
```

# Anonymous functions (2)

- A useful perspective in connection with currying:

instead of:

```
add :: Int → Int → Int
add x y = x + y
```

also:

```
add :: Int → (Int → Int)
add = \x → \y → x + y
```

- Or also:

```
const :: Int → Int → Int
const x _ = x
```

vs.

```
const :: Int → (Int → Int)
const x = \_ → x
```

- Also, abbreviating notation for anonymous functions of several arguments:

```
Main> (\x -> \y -> 2*x*y) 2 3
12
```

vs.

```
Main> (\x y -> 2*x*y) 2 3
12
```