



Functional Programming

Week 7 – Higher-Order Functions

René Thiemann Jonathan Bodemann James Fox Joshua Ocker Daniel Rainer
Daniel Ranalter Christian Sternagel

Department of Computer Science

Higher-Order Functions

Last Lecture

- type class definitions


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


```
instance (...) => TCName (TConstr a1 .. aN) where
  ... -- implementation of functions
```
- examples
 - classes: `Eq a`, `Num a`, `Integral a`, `RealFrac a`, ...
 - instances: `Integral Int`, `Eq a => Eq (Maybe a)`, `(Ord a, Ord b) => Ord (a,b)`, ...
- documentation:

<http://hackage.haskell.org/package/base-4.17.0.0/docs/Prelude.html>
- switch between operators and function names: `(+)` and ``div``

RT et al. (DCS @ UIBK)

Week 7

2/20

Functions and Values

- functions take values as input and produce output values
 - values so far: numbers, characters, pairs, lists, user defined datatypes, ...
 - examples
 - `lookup :: Eq a => a -> [(a,b)] -> Maybe b`
 - `elem :: Eq a => [a] -> Bool`
- important extension: **functions are values**
- result: **higher-order functions**
 - functions can take other functions as input, e.g.,


```
nTimes :: (a -> a) -> Int -> a -> a
-- nTimes f n x = f...(f x)
```
 - the result of a function can be a function, e.g.,


```
compose :: (b -> c) -> (a -> b) -> (a -> c)
-- compose f g is the function that takes an x and results in f(g(x))
```
- observations
 - higher-order functions are quite natural to define, e.g., `compose f g x = f (g x)`
 - higher-order functions are useful to avoid code duplication

Partial Application

- question: how to construct values that are functions?
- possible answer: **partial application**
- note: type constructor for functions (\rightarrow) associates to the right, cf. [lecture 4, slide 10](#)

$a \rightarrow b \rightarrow c \rightarrow d$ is identical to $a \rightarrow (b \rightarrow (c \rightarrow d))$

- note: function application associates to the left

$f \text{ expr1 expr2 expr3}$ is identical to $((f \text{ expr1}) \text{ expr2}) \text{ expr3}$

- example with parentheses added

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

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

- partial application: `average` is applied on less than two arguments

- example **expressions**

- `average :: Double -> (Double -> Double)` no arguments applied
- `average 3 :: Double -> Double` 1 argument applied
- `(average 3) 5 :: Double` first 1 argument applied, then another one
- `average 3 5 :: Double` same as above

Sections, flip

- **sections** are a special form of partial applications in combination with operators &

- `(expr &)` is the same as `(&) expr`

- `(& expr)` is a function that takes an `x` and returns `x & expr`

- `(& expr)` is the same as `flip (&) expr`

- `flip` is a predefined function that swaps the arguments of a binary function

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

```
-- same as (a -> b -> c) -> b -> a -> c
```

```
flip f y x = f x y
```

- exception: `(- expr)` is not `flip (-) expr` but just the negated value of `expr`

- examples

- `(> 3)`
- `(3 >)`
- `(3 -)`
- `(- 3)`

test whether a number is larger than 3
test whether 3 is larger than a number
subtract something from 3
the number -3

Example: nTimes

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

```
nTimes f n x
```

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

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

- observations

- `nTimes` uses standard recursion on numbers
- in the last line `f` is used twice
 - once as parameter of `nTimes`, where in `nTimes f` no argument is applied to `f`
 - once as the function which is applied to an argument: `otherwise = f (...)`

- application: implement other functions in more concise way

```
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
```

Partial Application and Evaluation

- if defining equation of `f` is of shape `f pat1 ... patN` with `N` arguments, then evaluation of `f expr1 ... exprM` can only happen, if $M \geq N$

- example `nTimes` and `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
```

```
-- (4 ^) cannot be evaluated!
```

```
= 4 ^ (nTimes (4 ^) 1 1)
```

```
-- evaluate second argument of ^
```

```
= 4 ^ (4 ^ (nTimes (4 ^) 0 1))
```

```
-- again, argument evaluation
```

```
= 4 ^ (4 ^ 1)
```

```
= 4 ^ 4
```

```
= 256
```

Partial Application and Evaluation, Continued

- if defining equation of `f` is of shape `f pat1 ... patN` with `N` arguments, then evaluation of `f expr1 ... exprM` can only happen, if $M \geq N$
- example with $M > N$

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

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

- restriction: all defining equations of a function must have same number of arguments
- consequence: the following code is not allowed, although it would make sense

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

Currying

- most of the time we defined functions in **curried form** (Haskell B. Curry, M. Schönfinkel)

```
f :: ty1 -> ... -> tyN -> ty
```

- alternative is **tupled form**

```
f :: (ty1, ..., tyN) -> ty
```

- observations

- partial application is only possible with curried form
- tupled form has advantage when passing logically connected values around

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

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

- argument order is relevant in curried form: partial application only possible from left to right
 - divide 1000 by something: `div 1000`
 - division by 1000: `let f x = div x 1000 in f`
 - alternative using `flip`: `flip div 1000`
- rule of thumb: put arguments that are unlikely to change to the left

Anonymous Functions: λ abstractions

- example: apply n -times the function that given an x computes $3 \cdot (x + 1)$
- one possibility: local definition of a function

```
example :: Num a => Int -> a -> a
example = let f x = 3 * (x + 1) in nTimes f
-- this is equivalent to
example n y = let f x = 3 * (x + 1) in nTimes f n y
```
- annoying: creation of function names, here `f`
- alternative: creation of anonymous function via **λ abstraction**
 - syntax: `\ pat1 ... patN -> expr` λ is written as `\` in Haskell
 - equivalent to: `let f pat1 ... patN = expr in f` for some fresh name `f`

```
example = nTimes (\ x -> 3 * (x + 1))
```
- difference between lambda abstractions and local function definitions
 - recursion not expressible via lambda abstractions
 - lambda abstractions do not require new function names

Example Higher-Order Functions and Applications

Generalize Common Programming Patterns

- consider the following tasks
 - multiply all list elements by 2
 - convert all characters in a string to upper case
 - compute a list of email addresses from a list of students
- possible implementation

```
multTwo [] = []
multTwo (x : xs) = 2 * x : multTwo xs
toUpperList [] = []
toUpperList (c : cs) = toUpper c : toUpperList cs
eMails [] = []
eMails (s : ss) = getEmail s : eMails ss
```
- observation: all of these functions are similar
- abstract version: apply some function on each list element
- aim: program the abstract version only once (will be a higher-order function), and then just instantiate this function for each task

The map Function

- map applies a function on each list element

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```
- solve tasks from previous slide easily

```
multTwo = map (2 *)
toUpperList = map toUpper
eMails = map getEmail
```
- example evaluation

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

The filter Function

- filter selects all elements of a list that satisfy some condition

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x : xs)
  | f x = x : filter f xs
  | otherwise = filter f xs
```
- example applications

```
-- test whether some element is included in a list
elem :: Eq a => a -> [a] -> Bool
elem x xs = filter (== x) xs /= []

-- the well known lookup function
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x xs = case filter (\ (k,_) -> x == k) xs of
  [] -> Nothing
  ((_,v) : _) -> Just v
```

Application: Quicksort

- quicksort is an efficient sorting algorithm
- main idea: partition a non-empty list into small and large elements and sort recursively
- straight-forward implementation

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = -- x is pivot element
  qsort (filter (<= x) xs) ++ [x] ++ qsort (filter (> x) xs)
```
- implementation might be tuned in several ways
 - use `partition :: (a -> Bool) -> [a] -> ([a], [a])` once instead of `filter` twice
 - parametrize order
 - `qsortBy :: (a -> a -> Bool) -> [a] -> [a]`
 - `qsort = qsortBy (<=)`
 - take `random` pivot element, cf. lecture Algorithms and Data Structures

The Function Composition Operator (.)

- function composition is a higher-order function (in Haskell: (..))
`(.) :: (b -> c) -> (a -> b) -> (a -> c)`
`(f . g) = \ x -> f (g x)`
- it takes two functions as input and returns a function
- in Haskell, function composition is often used to chain several function applications without explicit arguments
- example: given a number, first add 5, then compute the absolute value, then multiply it by 7, and finally convert it into a string and determine its length
- without composition: many parenthesis, not very readable
`\ x -> length (show ((abs (x + 5)) * 7))`
- written conveniently with function composition
`length . show . (* 7) . abs . (+ 5)`

Collection View

- often lists are used to encode collections of elements
- then one can process the whole collection via `map`, `filter`, `sum`, ... without looking at the position of the list elements
- list index function (`!!`) is rarely used in these applications
- in particular: do **not** write the following kind of loop

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

as functional program
`map (\ i -> someFun (xs !! i)) [0 .. length xs - 1]`but instead just write
`map someFun xs`
- the bad program needs $\sim \frac{1}{2}n^2$ evaluation steps for a list of length n : **lists \neq arrays!**

Application: Names of Good Students

- given a list of students, compute a sorted list of all names of students whose average grade is 2 or better
- implementation

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

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

Summary

- higher-order functions
 - functions may have functions as input
 - functions may have functions as output
- partial application
 - n -ary function is value
 - applying n -ary function on 1 argument results in $n - 1$ -ary function
 - sections are special syntax for partially applied operators
- λ -abstraction is anonymous function
- process lists that encode a collection via `map`, `filter`, ...