

Functional Programming



http://www.fh-nw.ch/nwred/fh/red/1000x600/download/d/15792

Functions II

Daniel Kröni

University of Applied Sciences Northwestern Switzerland

(C) Hochschule für Technik
Fachhochschule Nordwestschweiz

Learning Targets

You

- know the concept of currying
- understand how partial application relates to currying
- are able to apply lambda expressions
- know how to define and use operators
- can compose functions

Content

- **Currying / Partial Application**
- **Lambda Expressions**
- **Function Composition**
- **Operators**



Functions with Multiple Parameters

- **Every function in Haskell takes only one parameter**

```
add :: Int -> Int -> Int  
add a b = a + b
```

- **The function type arrow ' \rightarrow ' associates to the right**

add :: (Int -> (Int -> Int))	✓	add :: ((Int -> Int) -> Int)	✗
------------------------------	---	------------------------------	---

- The function add takes a single **Int** parameter and returns a function which accepts another **Int** parameter and which finally returns a result of type **Int**

- **Function application associates to the left**

add 1 2 == ((add 1) 2)	✓	(add (1 2))	✗
------------------------	---	-------------	---

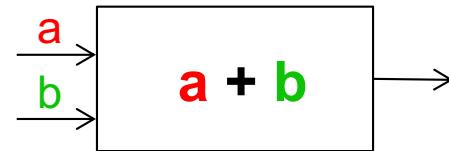
- **These two rules allow for concise function application syntax**



Partial Application

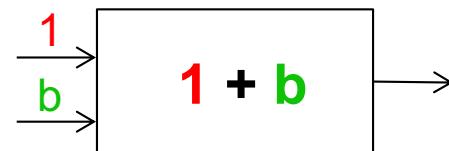
- Applying add to zero arguments

```
add :: Int -> Int -> Int  
add a b = a + b
```



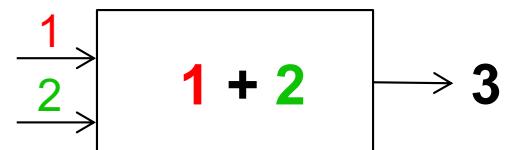
- Applying add to the first argument

```
add 1 :: Int -> Int
```



- Applying add to both arguments

```
(add 1) 2 :: Int
```



- Functions are values and can be bound to a name

```
inc :: Int -> Int  
inc = add 1
```

Worksheet: Currying



Lambda Expressions

- **The following function definition does two things**

```
inc x = x + 1
```

- It constructs a function: The function which adds one to its argument
- It binds the name 'inc' to this function
- **These two things are conceptually independent of each other**
- **Many reasons to bind names to values (functions are values btw.)**
 - The value is referred to more than once
 - The value is recursively used in its own declaration
 - The value needs a type signature

Lambda Expressions

- **But not everything needs a name**

```
inc x = x + one  
one = 1
```

- There is no good reason for naming the value 1

- **What about functions which are used only once**

```
incAll :: [Int] -> [Int]  
incAll xs = map inc xs  
  
inc :: Int -> Int  
inc x = x + 1
```

- There is no good reason for naming the increment function

- **We want the functionality without the name**

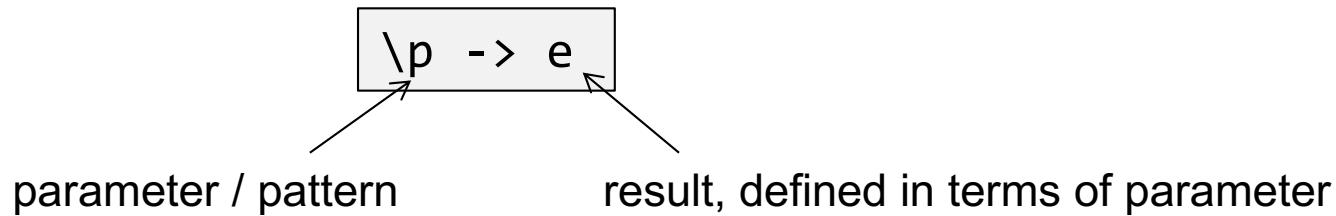
```
incAll :: [Int] -> [Int]  
incAll xs = map (\x -> x + 1) xs
```

Lambda Expressions

- Functions can be defined anonymously

```
\x -> x + 1
```

- This is called a **lambda expression**
- General form



- Translation

```
\p1 ... pn -> e
```

==

```
\x1 -> ... -> \xn =  
  case (x1, ..., xn)  
    of (p1, ..., pn) -> e
```

Lambda Expressions and Currying

- All four functions have the same type

```
prod1, prod2, prod3, prod4 :: Int -> Int -> Int -> Int
```

```
prod1 x y z = x * y * z
```

```
prod2 x y     = \z -> x * y * z
```

```
prod3 x        = \y -> \z -> x * y * z
```

```
prod4          = \x -> \y -> \z -> x * y * z
```

- Actually the definitions of prod1, prod2 and prod3 all get translated to prod4 by the compiler.

Lambda Expressions in other Languages



```
Arrays.asList(1,2,3).stream()  
.map(i-> i+1)  
.collect(Collectors.toList());
```



```
list(map(lambda i: i+1, [1,2,3]))
```



```
[1,2,3].map(i => i+1)
```



```
using System.Linq;  
new List<int>() { 1,2,3 }.Select(i => i+1)
```

Worksheet: Lambda Expressions



HOFs : Higher-Order Functions

- A function can return another function as its result

```
add :: Int -> Int -> Int == add :: Int -> (Int -> Int)
```

- A function can also take another function as its argument

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

- First parameter is a function from $a \rightarrow \text{Bool}$

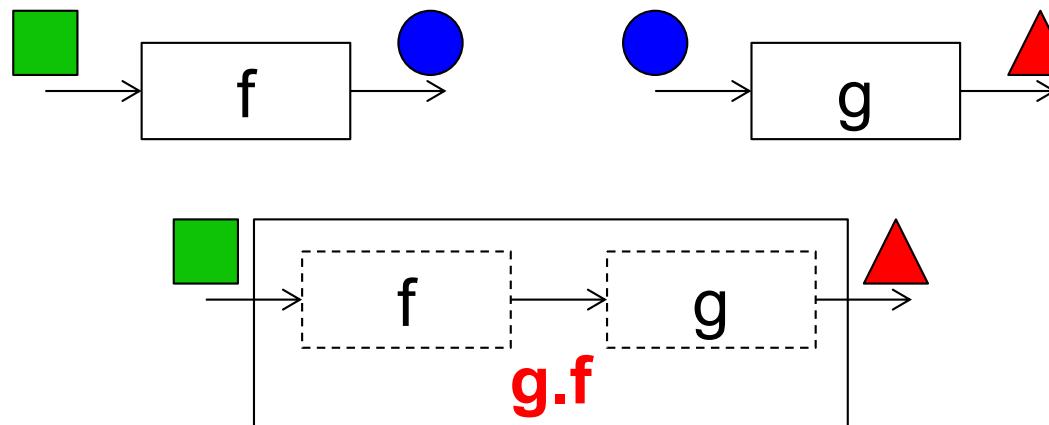
- Functions are just values! They can be

- input and output to functions
- put into and retrieved from data structures

- Functions that have functions as input and/or output are called **higher-order functions**

Function Composition: Pipelines of functions

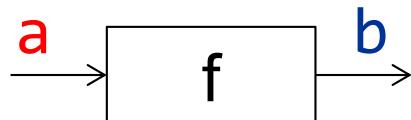
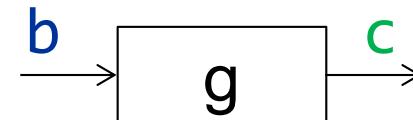
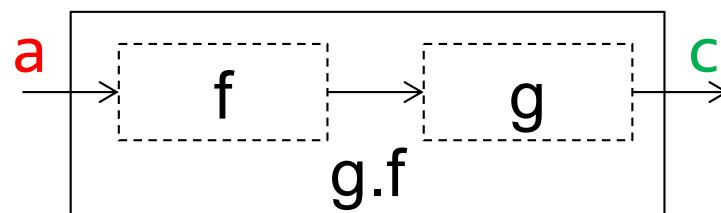
- Functions can be combined into pipelines if the types line up



- Function composition of two functions takes the output of one function as the input of a second one.
- The argument order originates from mathematics $g(f(x))$
 - First apply f to x and then apply g to the result: g followed by f

Function Composition: Pipelines of functions

- Functions can be combined into pipelines if the types line up

$$f :: a \rightarrow b$$

$$g :: b \rightarrow c$$

$$\begin{aligned}(_) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ g . f &= \lambda x \rightarrow g(f x)\end{aligned}$$


```
map (\xs -> negate (sum (tail xs))) [[1,2,3],[4,5,6],[7,8]]
```

```
map (negate . sum . tail) [[1,2,3],[4,5,6],[7,8]]
```

Operators

- **Function names start with a lower case letter ['a'..'z']**
 - Function names are written in front of their arguments (prefix)

```
add :: Int -> Int -> Int
add a b = a + b
```

```
add 1 2 ~> 3
```

- **Operator symbols are formed from one or more symbol characters !#\$%&*+../<=>?@\\^|-~** [Haskell 2010 Report §2.4](#)
 - Operator symbols are written between its arguments (infix)

```
(|+|) :: Int -> Int -> Int
a |+| b = abs a + abs b
```

```
1 |+| (-2) ~> 3
```

Operators

- An Operator can be used as a function by wrapping it with parentheses (`op`)

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> (+) 1 2
3
```

- A function can be used as an operator by quoting it with backticks `fun` (not to be confused with ')

```
Prelude> :t div
div :: Integral a => a -> a -> a
Prelude> 9 `div` 2
4
```

- The **syntax** changes but the **semantics** (meaning) remains the same

Sections

- **Infix operators can be partially applied as well.**
 - This is called a section
 - The parentheses are mandatory

$(2+)$ is interpreted as $\lambda y \rightarrow 2+y$
 $(+3)$ is interpreted as $\lambda x \rightarrow x+3$
- **If the operator is applied to no argument at all haskell essentially treats an infix operator as an equivalent functional value.**

$(+)$ = $\lambda x y \rightarrow x+y$

Operator Precedence

- Operators have a defined precedence

`1 + 2 * 3 == 1 + (2 * 3)`



`not: (1 + 2) * 3`



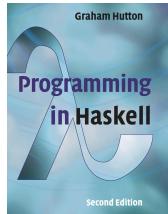
- This can be defined using a fixity declaration

- Declares a precedence level from 0 to 9 (with 9 being the strongest)
 - Normal function application is assumed to have a precedence level of 10
- Specifies the operator to be
 - left-associative: infixl right-associative: infixr
 - non-associative: infix

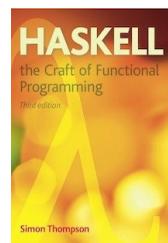
```
Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
  ...
infixl 6 +
```

```
Prelude> :info (*)
class Num a where
  (*) :: a -> a -> a
  ...
infixl 7 *
```

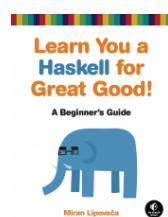
Further Reading



Chapters:
3.6, 4.5, 4.6, 7.5



Chapter 11
Pages 231 - 252



Chapter 5: Higher order functions
<http://learnyouahaskell.com/higher-order-functions>