

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 2



Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Types and Type Classes
- Lists
- Patterns, Guards, and More
- Higher-Order Functions

Types and Type Classes

Basic Concepts

- **types** τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with **type variables** $\alpha - \mathbf{a}, \mathbf{b}, \dots$

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables $\alpha - \mathbf{a}, \mathbf{b}, \dots$
- **type constructors** $C - \mathbf{Bool}, \mathbf{Int}, [], (,), \dots$

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables $\alpha - a, b, \dots$
- type constructors $C - \text{Bool}, \text{Int}, [], (,), \dots$

think List

think Pair

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables α – **a**, **b**, ...
- type constructors C – **Bool**, **Int**, **[]**, **(,)**, ...
- **function type constructor** \rightarrow (special case of previous item)

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables α – **a**, **b**, ...
- type constructors C – **Bool**, **Int**, **[]**, **(,)**, ...
- function type constructor \rightarrow (special case of previous item)
- \rightarrow associates to the right: $\tau \rightarrow (\tau \rightarrow \tau) = \tau \rightarrow \tau \rightarrow \tau$

Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables α – **a**, **b**, ...
- type constructors C – **Bool**, **Int**, **[]**, **(,)**, ...
- function type constructor \rightarrow (special case of previous item)
- \rightarrow associates to the right: $\tau \rightarrow (\tau \rightarrow \tau) = \tau \rightarrow \tau \rightarrow \tau$
- types denote collections of related values, e.g.,
Bool = {**True**, **False**}

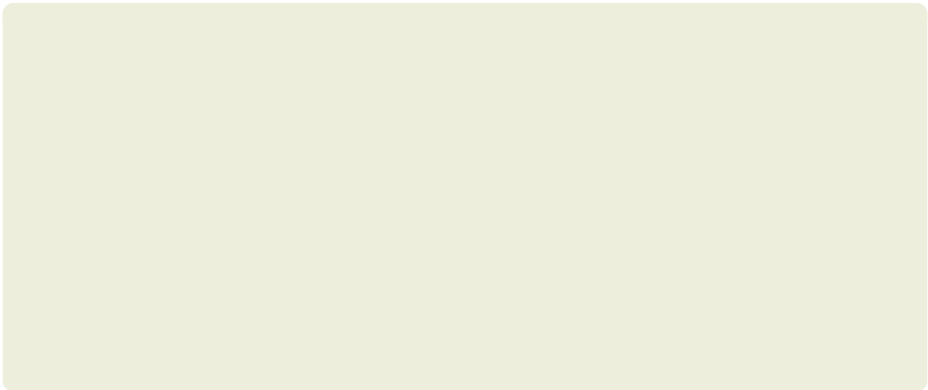
Basic Concepts

- types τ are built according to grammar

$$\tau \stackrel{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with type variables α – **a**, **b**, ...
- type constructors C – **Bool**, **Int**, **[]**, **(,)**, ...
- function type constructor \rightarrow (special case of previous item)
- \rightarrow associates to the right: $\tau \rightarrow (\tau \rightarrow \tau) = \tau \rightarrow \tau \rightarrow \tau$
- types denote collections of related values, e.g.,
Bool = {**True**, **False**}
- **type signature/constraint** $e :: \tau$ means “ e is of type τ ”

Basic Types



Basic Types

- `Bool` – logical values (`True`, `False`)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)

Basic Types

- **Bool** – logical values (**True**, **False**)
- **Char** – single characters ('a', '\n', ...)
- **String** – sequences of characters ("abc", "1+2=3")
- **Int** – fixed-precision integers between -2^{31} and $2^{31} - 1$ (-100, 0, 999)

Basic Types

- **Bool** – logical values (**True**, **False**)
- **Char** – single characters ('a', '\n', ...)
- **String** – sequences of characters ("abc", "1+2=3")
- **Int** – fixed-precision integers between -2^{31} and $2^{31} - 1$ (-100, 0, 999)
- **Integer** – arbitrary-precision integers

Basic Types

- **Bool** – logical values (**True**, **False**)
- **Char** – single characters ('a', '\n', ...)
- **String** – sequences of characters ("abc", "1+2=3")
- **Int** – fixed-precision integers between -2^{31} and $2^{31} - 1$ (**-100**, **0**, **999**)
- **Integer** – arbitrary-precision integers
- **Float** – single-precision floating-point numbers (**-12.34**, **1.0**, **3.14159**)

Basic Types

- **Bool** – logical values (**True**, **False**)
- **Char** – single characters ('a', '\n', ...)
- **String** – sequences of characters ("abc", "1+2=3")
- **Int** – fixed-precision integers between -2^{31} and $2^{31} - 1$ (-100, 0, 999)
- **Integer** – arbitrary-precision integers
- **Float** – single-precision floating-point numbers (-12.34, 1.0, 3.14159)
- **Double** – double-precision floating-point numbers

Basic Types

- **Bool** – logical values (`True`, `False`)
- **Char** – single characters (`'a'`, `'\n'`, ...)
- **String** – sequences of characters (`"abc"`, `"1+2=3"`)
- **Int** – fixed-precision integers between -2^{31} and $2^{31} - 1$ (`-100`, `0`, `999`)
- **Integer** – arbitrary-precision integers
- **Float** – single-precision floating-point numbers (`-12.34`, `1.0`, `3.14159`)
- **Double** – double-precision floating-point numbers

Note – Show Types in GHCi

- `Prelude> :set +t`
- commands may be put inside `~/ghci` (read on GHCi startup)

List Types

- type of lists with elements of type τ : $[\tau]$
- all elements are of same type
- no restriction on length of list

List Types

- type of lists with elements of type τ : $[\tau]$
- all elements are of same type
- no restriction on length of list

Tuple Types

- type of tuples with elements of types τ_1, \dots, τ_n : (τ_1, \dots, τ_n)
- length: 2 (pair), 3 (triple), 4 (quadruple), \dots , n (n -tuple), \dots
- elements may be of different types
- fixed number of elements

List Types

- type of lists with elements of type τ : $[\tau]$
- all elements are of same type
- no restriction on length of list

Tuple Types

- type of tuples with elements of types τ_1, \dots, τ_n : (τ_1, \dots, τ_n)
- length: 2 (pair), 3 (triple), 4 (quadruple), \dots , n (n -tuple), \dots
- elements may be of different types
- fixed number of elements

Examples

```
['a','b','c','d'] :: [Char]
["One","Two","Three"] :: [String]
[['a','b'],['c','d','e']] :: [[Char]]
(False,True) :: (Bool,Bool)
("Yes",True,'a') :: (String,Bool,Char)
```

Function Types

- type of functions from values of type τ_1 to values of type τ_2 : $\tau_1 \rightarrow \tau_2$
- every function takes single argument and returns single result
- simulating multiple arguments: use tuples

Function Types

- type of functions from values of type τ_1 to values of type τ_2 : $\tau_1 \rightarrow \tau_2$
- **every function** takes **single** argument and returns **single** result
- simulating multiple arguments: use tuples

Function Types

- type of functions from values of type τ_1 to values of type τ_2 : $\tau_1 \rightarrow \tau_2$
- every function takes single argument and returns single result
- simulating multiple arguments: use tuples

Examples

```
not :: Bool -> Bool
```

```
not True = False
```

```
not False = True
```

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

```
add (x, y) = x + y
```

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of **partial application**, curried functions are more versatile than uncurried functions

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Anonymous Functions – “Lambda-Abstractions”

- $\lambda x \rightarrow e$ is function taking x and returning e

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Anonymous Functions – “Lambda-Abstractions”

- $\lambda x \rightarrow e$ is function taking x and returning e

Example

```
add' = \x -> \y -> x + y
```

Basic Functions

- `Bool`: conjunction `&&`, disjunction `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`

Basic Functions

- `Bool`: conjunction `&&`, disjunction `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`
- `(a, b)`: choose first `fst`, choose second `snd`

Basic Functions

- `Bool`: conjunction `&&`, disjunction `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`
- `(a, b)`: choose first `fst`, choose second `snd`

Examples

```
not True      == False
(False && x)  == False
(True || x)   == True
otherwise     == True
```

```
fst (x, y) == x
snd (x, y) == y
```

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: $e :: C \ a \Rightarrow \tau$
- meaning: “for every type a of class C , the type of e is τ ” (where τ does contain a)

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: $e :: C \ a \Rightarrow \tau$
- meaning: “for every type a of class C , the type of e is τ ” (where τ does contain a)

Example – Addition

- $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
- “for every type a of class Num , addition has type $a \rightarrow a \rightarrow a$ ”
- since, e.g., Int is of class Num , we obtain that addition is of type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$, when used on Ints

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: $e :: C \ a \Rightarrow \tau$
- meaning: “for every type a of class C , the type of e is τ ” (where τ does contain a)

Example – Addition

(op) turns infix op into prefix

- $(+)$ $:: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
- “for every type a of class Num , addition has type $a \rightarrow a \rightarrow a$ ”
- since, e.g., Int is of class Num , we obtain that addition is of type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$, when used on Ints

The Eq Class – Equality

- specification, one of:

`(==)` :: Eq a => a -> a -> Bool

`(/=)` :: Eq a => a -> a -> Bool

The Eq Class – Equality

- specification, one of:
`(==) :: Eq a => a -> a -> Bool`
`(/=) :: Eq a => a -> a -> Bool`

The Ord Class – Orders

- prerequisite: `Eq`
- specification, one of:
`compare :: Ord a => a -> a -> Ordering`
`(<=) :: Ord a => a -> a -> Bool`
- where `Ordering = {LT, EQ, GT}`
- additional functions: `(<)`, `(>=)`, `(>)`, `min`, `max`

The Eq Class – Equality

- specification, one of:
`(==)` :: `Eq a => a -> a -> Bool`
`(/=)` :: `Eq a => a -> a -> Bool`

The Ord Class – Orders

- prerequisite: `Eq`
- specification, one of:
`compare` :: `Ord a => a -> a -> Ordering`
`(<=)` :: `Ord a => a -> a -> Bool`
- where `Ordering = {LT, EQ, GT}`
- additional functions: `(<)`, `(>=)`, `(>)`, `min`, `max`

The Read Class – “from string”

- useful functions:
`read` :: `Read a => String -> a`

The Show Class – “to string”

- specification, one of:

```
show      :: Show a => a -> String
```

```
showsPrec :: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The Show Class – “to string”

- specification, one of:

```
show      :: Show a => a -> String
```

```
showsPrec :: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The Num Class – Numeric Types

- prerequisites: `Eq` and `Show`

- specification, all of:

```
(+)      :: Num a => a -> a -> a
```

```
(*)      :: Num a => a -> a -> a
```

```
(-)      :: Num a => a -> a -> a
```

```
abs      :: Num a => a -> a
```

```
signum   :: Num a => a -> a
```

```
fromInteger :: Num a => Integer -> a
```

- additional functions: `negate`

The Show Class – “to string”

- specification, one of:

```
show      :: Show a => a -> String
```

```
showsPrec :: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The Num Class – Numeric Types

- prerequisites: `Eq` and `Show`

- specification, all of:

```
(+)      :: Num a => a -> a -> a
```

```
(*)      :: Num a => a -> a -> a
```

```
(-)      :: Num a => a -> a -> a
```

```
abs      :: Num a => a -> a
```

```
signum   :: Num a => a -> a
```

```
fromInteger :: Num a => Integer -> a
```

- additional functions: `negate`

visit: <http://haskell.org> → Documentation → Language Report: Haskell 2010

Lists

Constructing Lists

- $[a] \stackrel{\text{def}}{=} [] \mid a : [a]$
- for given list, exactly two cases: either empty $[]$, or contains at least one element x and a remaining list xs ($x : xs$)
- $[x_1, x_2, \dots, x_n]$ abbreviates $x_1 : (x_2 : (\dots : (x_n : []) \dots))$
- $(:)$ is right-associative, hence $x_1 : (x_2 : xs) = x_1 : x_2 : xs$

Constructing Lists

- $[a] \stackrel{\text{def}}{=} [] \mid a : [a]$
- for given list, exactly two cases: either empty $[]$, or contains at least one element x and a remaining list xs ($x : xs$)
- $[x_1, x_2, \dots, x_n]$ abbreviates $x_1 : (x_2 : (\dots : (x_n : []) \dots))$
- $(:)$ is right-associative, hence $x_1 : (x_2 : xs) = x_1 : x_2 : xs$

Examples

$$\begin{aligned} 1 : (2 : (3 : (4 : []))) &== 1 : 2 : 3 : 4 : [] \\ 1 : 2 : 3 : 4 : [] &== [1, 2, 3, 4] \\ 1 : [2, 3, 4] &== [1, 2, 3, 4] \end{aligned}$$

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- polymorphic means “having many forms”
- definition

```
myReplicate n x =  
  if n <= 0 then []  
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord n, Num n) => n -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `n`

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- **polymorphic** means “having many forms”
- definition

```
myReplicate n x =  
  if n <= 0 then []  
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord n, Num n) => n -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `n`

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- polymorphic means “having many forms”
- definition

```
myReplicate n x =  
  if n <= 0 then []  
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord n, Num n) => n -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `n`

Exercise

use equational reasoning to evaluate `myReplicate 2 'c'`

Testing for Emptiness

- `null :: [a] -> Bool - True` iff argument is empty list

Testing for Emptiness

- `null :: [a] -> Bool - True` iff argument is empty list

Functions on Integer Lists

```
range m n =
  if m > n then []
  else m : range (m + 1) n

mySum xs =
  if null xs then 0
  else head xs + mySum (tail xs)

prod xs =
  if null xs then 1
  else head xs * prod (tail xs)
```

Examples

```
range 1 3 = [1,2,3]
```

```
range 3 2 = []
```

```
mySum [1,2,3] = 1 + 2 + 3 + 0
```

```
mySum [] = 0
```

```
prod [1,2,3] = 1 * 2 * 3 * 1
```

```
prod [] = 1
```

$$\text{mySum (range 1 n)} = \sum_{i=1}^n i$$

Patterns, Guards, and More

Patterns

- used to match specific cases

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} -$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} _$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

- $_$ matches everything and ignores the result

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} _$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

- $_$ matches everything and ignores the result
- x matches everything and binds the result to x

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} _$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

- $_$ matches everything and ignores the result
- x matches everything and binds the result to x
- $x @ \langle pat \rangle$ matches the same as $\langle pat \rangle$ and binds result to x

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} _$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

- $_$ matches everything and ignores the result
- x matches everything and binds the result to x
- $x @ \langle pat \rangle$ matches the same as $\langle pat \rangle$ and binds result to x
- constructor patterns match the described application of a data constructor (example constructors: $(:)$ and $[]$ for lists, **True** and **False** for Boolean values, ...)

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	$\stackrel{\text{def}}{=} _$	wildcard
	x	variable pattern
	$x @ \langle pat \rangle$	“as” pattern
	$[\langle pat \rangle, \dots, \langle pat \rangle]$	list pattern
	$(\langle pat \rangle, \dots, \langle pat \rangle)$	tuple pattern
	$C \langle pat \rangle \dots \langle pat \rangle$	constructor pattern

- $_$ matches everything and ignores the result
- x matches everything and binds the result to x
- $x @ \langle pat \rangle$ matches the same as $\langle pat \rangle$ and binds result to x
- constructor patterns match the described application of a data constructor (example constructors: $(:)$ and $[]$ for lists, **True** and **False** for Boolean values, ...)
- patterns may be used in arguments of function definitions and together with the **case**-construct

The case Construct

$$\begin{array}{l} \text{case } e \text{ of } \langle pat_1 \rangle \rightarrow e_1 \\ \quad \quad \quad \vdots \\ \quad \quad \quad \langle pat_n \rangle \rightarrow e_n \end{array}$$

- checks $\langle pat_1 \rangle$ to $\langle pat_n \rangle$ top to bottom
- if $\langle pat_i \rangle$ is first match, e_i is evaluated

The case Construct

```
case e of ⟨pat1⟩ -> e1
         ⋮
         ⟨patn⟩ -> en
```

- checks $\langle pat_1 \rangle$ to $\langle pat_n \rangle$ top to bottom
- if $\langle pat_i \rangle$ is first match, e_i is evaluated

Example – Pattern Matching

```
mySum [] = ... -- constructor pattern
fst (x, _) = x -- patterns: tuple, variable, wildcard
case xs of [x] -> ... -- patterns: list, variable
           _   -> ... -- wildcard
```

Pattern Guards

- a pattern may be followed by a guard b

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle pat \rangle \mid b$

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle pat \rangle \mid b$
- where b is a Boolean expression

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle pat \rangle \mid b$
- where b is a Boolean expression

Example

```
f1 (x, _) | x >= 0 = x -- only if x non-negative
f2 (x:xs) | null xs = ... -- same as [x]
```

Refined Definitions

```
head (x:_) = x
```

```
tail (_:xs) = xs
```

```
myReplicate n x | n <= 0    = []  
                | otherwise = x : myReplicate (n - 1) x
```

```
null [] = True
```

```
null _ = False
```

```
range m n | m > n    = []  
          | otherwise = m : range (m + 1) n
```

```
mySum [] = 0
```

```
mySum (x:xs) = x + mySum xs
```

```
prod [] = 1
```

```
prod (x:xs) = x * prod xs
```

Higher-Order Functions

Definition

a function is of **higher-order** if

- it takes functions as arguments and/or
- returns function as result

Definition

a function is of higher-order if

- it takes functions as arguments and/or
- returns function as result

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Definition

a function is of higher-order if

- it takes functions as arguments and/or
- returns function as result

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Sections

- abbreviation for partially applied infix operators
- $(x \text{ `op` } y)$ abbreviates $(\backslash y \rightarrow x \text{ `op` } y)$
- $(\text{ `op` } y)$ abbreviates $(\backslash x \rightarrow x \text{ `op` } y)$

Definition

a function is of higher-order if

- it takes functions as arguments and/or
- returns function as result

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Sections

- abbreviation for partially applied infix operators
- $(x \text{ `op` } y)$ abbreviates $(\backslash y \rightarrow x \text{ `op` } y)$
- $(\text{ `op` } y)$ abbreviates $(\backslash x \rightarrow x \text{ `op` } y)$

Examples

```
ghci> twice (^2) 10  
10000
```

Processing Lists – map

- possible definition

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

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

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

- syntactic sugar `map f xs = [f x | x <- xs]`

Processing Lists – map

- possible definition

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

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

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

- syntactic sugar `map f xs = [f x | x <- xs]`

Examples

```
ghci> map (+1) [1,3,5,7]
```

```
[2,4,6,8]
```

```
ghci> import Data.Char
```

```
ghci> map isDigit ['a','1','b','2']
```

```
[False,True,False,True]
```

```
ghci> map reverse ["abc","def","ghi"]
```

```
["cba","fed","ihg"]
```

```
ghci> map (map (+1)) [[1,2,3],[4,5]]
```

```
[[2,3,4],[5,6]]
```

Processing Lists – filter

- possible definition

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

```
filter p [] = []
```

```
filter p (x:xs)
```

```
  | p x          = x : filter p xs
```

```
  | otherwise    = filter p xs
```

- syntactic sugar `filter p xs = [x | x <- xs, p x]`

Processing Lists – filter

- possible definition

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

```
filter p [] = []
```

```
filter p (x:xs)
```

```
  | p x          = x : filter p xs
```

```
  | otherwise    = filter p xs
```

- syntactic sugar `filter p xs = [x | x <- xs, p x]`

Examples

```
ghci> filter even [1..10]
```

```
[2,4,6,8,10]
```

```
ghci> filter (>5) [1..10]
```

```
[6,7,8,9,10]
```

```
ghci> filter (/= ' ') "abc def ghi"
```

```
"abcdefghi"
```


“Fold Right” – A Very Expressive Function

- possible definition

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

```
foldr f b [] = b
```

```
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- b is ‘base value’
- f combining function (binary)
- intuitively $\text{foldr } f \ b \ [x_1, x_2, \dots, x_n]$

$$\begin{aligned} &= \text{foldr } f \ b \ (x_1 : (x_2 : \dots (x_n : []) \dots)) \\ &= (x_1 \text{ `f` } (x_2 \text{ `f` } \dots (x_n \text{ `f` } b) \dots)) \end{aligned}$$

“Fold Right” – A Very Expressive Function

- possible definition

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

```
foldr f b [] = b
```

```
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- `b` is ‘base value’
- `f` combining function (binary)
- intuitively `foldr f b [x1, x2, ..., xn]`

$$\begin{aligned} &= \text{foldr } f \text{ } b \text{ } (x_1 : (x_2 : \dots (x_n : []) \dots)) \\ &= (x_1 \text{ `f` } (x_2 \text{ `f` } \dots (x_n \text{ `f` } b) \dots)) \end{aligned}$$

This Pattern is Very General

- take (+) for `f` and 0 for `b`: `foldr (+) 0 = sum`
- take (*) for `f` and 1 for `b`: `foldr (*) 1 = product`
- take `const (+1)` for `f` and 0 for `b`:
`foldr (const (+1)) 0 = length` (where `const f _ = f`)

“Fold Right” – A Very Expressive Function

- possible definition

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

```
foldr f b [] = b
```

```
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- `b` is ‘base value’
- `f` combining function (binary)
- intuitively `foldr f b [x1, x2, ..., xn]`

$$= \text{foldr } f \text{ } b \text{ } (x_1 : (x_2 : \dots (x_n : []) \dots))$$
$$= (x_1 \text{ `f` } (x_2 \text{ `f` } \dots (x_n \text{ `f` } b) \dots))$$

This Pattern is Very General

- take (+) for `f` and 0 for `b`: `foldr (+) 0 = sum`
- take (*) for `f` and 1 for `b`: `foldr (*) 1 = product`
- take `const (+1)` for `f` and 0 for `b`:

```
foldr (const (+1)) 0 = length (where const f _ = f)
```

add dummy argument

Exercises (for October 27th)

1. Read Chapters 1 and 2 of [Real World Haskell](#).
2. Work through lessons 4 to 5 on <http://tryhaskell.org/>.
3. Give the types (and class constraints) for each of:

`second xs = head (tail xs)`

`swap (x, y) = (y, x)`

`pair x y = (x, y)`

`double x = x * 2`

`palindrome xs = reverse xs == xs`

`twice f x = f (f x)`

4. Use equational reasoning to compute the result of `filter (== 3) [1,2,3]` (on paper). Give all intermediate steps.
5. Using `foldr`, give alternative definitions of two of the functions we have seen so far (not including those that have already been defined via `foldr`).
6. Define a function `concat :: [[a]] -> [a]` that concatenates a list of lists, e.g., `concat [[1], [], [2,3]] = [1,2,3]`.