Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

October 13, 2010

Today's Topics

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

Types and Classes

Basic Concepts

- types are built according to the grammar

$$\tau \stackrel{\mathit{def}}{=} \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ldots\ \tau$$

- where α is a type variable (like `a`, `b`, ...)
- and C a type constructor (like `Bool`, `Int`, `[]`, `(,)`)
- `->` associates to the right: $\tau \texttt{ -> } (\tau \texttt{ -> } \tau) = \tau \texttt{ -> } \tau \texttt{ -> } \tau$
- types denote collections of related values, e.g., $\texttt{Bool} = \{\texttt{True}, \texttt{False}\}$
- e `::` τ means "e is of type τ"

Basic Concepts

- types are built according to the grammar

$$\tau \stackrel{\textit{def}}{=} \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ldots\ \tau$$

- where α is a type variable (like `a`, `b`, ...)
- and C a type constructor (like `Bool`, `Int`, `[]`, `(,)`)
- `->` associates to the right: $\tau \texttt{ -> } (\tau \texttt{ -> } \tau) = \tau \texttt{ -> } \tau \texttt{ -> } \tau$
- types denote collections of related values, e.g., $\texttt{Bool} = \{\texttt{True}, \texttt{False}\}$
- e `::` τ means "e is of type τ"

Basic Concepts

- types are built according to the grammar

$$\tau \stackrel{\mathit{def}}{=} \alpha \mid \tau \; \texttt{->} \; \tau \mid C \; \tau \; \ldots \; \tau$$

- where α is a type variable (like `a`, `b`, ...)
- and C a type constructor (like `Bool`, `Int`, `[]`, `(,)`)
- `->` associates to the right: $\tau \; \texttt{->} \; (\tau \; \texttt{->} \; \tau) = \tau \; \texttt{->} \; \tau \; \texttt{->} \; \tau$
- types denote collections of related values, e.g., $\texttt{Bool} = \{\texttt{True}, \texttt{False}\}$
- e `::` τ 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
  ```
- commonly used commands may be put inside `~/.ghci` (read on GHCi startup)
  ```

List Types

- type of lists with elements of type τ: `[`τ`]`
- 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 $\tau_1, \ldots, \tau_n$: $(\tau_1, \ldots, \tau_n)$
- length: 2 (pair), 3 (triple), 4 (quadruple), …, n (n-tuple), …
- 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 $\tau_1, \ldots, \tau_n$: $(\tau_1, \ldots, \tau_n)$
- length: 2 (pair), 3 (triple), 4 (quadruple), …, n (n-tuple), …
- 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)
(False,'a',True) :: (Bool,Char,Bool)
("Yes",True,'a') :: (String,Bool,Char)
```

Function Types

- τ_1 `->` τ_2 is type of all functions from inputs of type τ_1 to outputs of type τ_2
- every function takes single argument and returns single value
- simulating multiple arguments: use tuples

Function Types

- τ_1 `->` τ_2 is type of all functions from inputs of type τ_1 to outputs of type τ_2
- every function takes single argument and returns single value
- simulating multiple arguments: use tuples

Function Types

- τ_1 `->` τ_2 is type of all functions from inputs of type τ_1 to outputs of type τ_2
- every function takes single argument and returns single value
- simulating multiple arguments: use tuples

Examples

```
not :: Bool -> Bool

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

“Schönfinkelization”

- 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:  a function adding 10
add10 = add' 10
```

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

```haskell
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application:  a function adding 10
add10 = add' 10
```

Anonymous Functions - “Lambda-Abstractions”

- `\`x `->` 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

```haskell
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application:  a function adding 10
add10 = add' 10
```

Anonymous Functions - "Lambda-Abstractions"

- `\x -> e` is function taking x and returning e

Example

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

Basic Functions

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

Basic Functions

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

Basic Functions

- `Bool`: conjunction `(&&)`, disjunction `(||)`, negation `not`, 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 a standard set of operations
- use same name, independent of actual type

Overloaded Types

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

Realization - Class Constrains

- syntax: e `::` C `a => ` τ
- meaning: "for every type `a` of class C, the type of e is τ" (where τ does contain `a`)

Overloaded Types

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

Realization - Class Constrains

- syntax: e `::` C `a =>` τ
- meaning: "for every type `a` of class C, the type of e is τ" (where τ does contain `a`)

Example - Addition

- `(+) :: Num a => a -> a -> a`
- "for every type `a` of class `Num`, addition has type `a -> a -> a`"
- since, e.g., `Int` is of class `Num`, we obtain that addition is of type `Int -> Int -> Int`, when used on `Int`s

Overloaded Types

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

Realization - Class Constrains

- syntax: e `::` C `a =>` τ
- 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

- `(+) :: Num a => a -> a -> a`
- “for every type `a` of class `Num`, addition has type `a -> a -> a`”
- since, e.g., `Int` is of class `Num`, we obtain that addition is of type `Int -> Int -> Int`, when used on `Int`s

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` → `Standard libraries` → `Haskell 98 Prelude`

Lists

Constructing Lists

- `[a]` $\stackrel{def}{=}$ `[]` | `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`, ... ,`x_n`]` abbreviates x_1 `:` (x_2 `:` ($\cdots$ `:` (x_n `:` `[]`) $\cdots$))
- `(:)` is right-associative, hence x_1 `:` (x_2 `:` xs) $=$ x_1 `:` x_2 `:` xs

Constructing Lists

- `[a]` $\stackrel{def}{=}$ `[]` | `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`, ... ,`x_n`]` abbreviates x_1 : (x_2 : ($\cdots$: (x_n : `[]`)$\cdots$))
- `(:)` is right-associative, hence $x_1 : (x_2 : xs) = x_1 : x_2 : xs$

Examples

```
1 : (2 : (3 : (4 : []))) == 1 : 2 : 3 : 4 : []
1 : 2 : 3 : 4 : []       == [1,2,3,4]
1 : [2,3,4]              == [1,2,3,4]
```

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 `Int -> a -> [a]`, i.e., it can construct lists of arbitrary type `a`

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 `Int -> a -> [a]`, i.e., it can construct lists of arbitrary type `a`

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 `Int -> a -> [a]`, i.e., it can construct lists of arbitrary type `a`

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

$$
\begin{aligned}
\texttt{range 1 3} &= \texttt{[1,2,3]} \\
\texttt{range 3 2} &= \texttt{[]}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{mySum [1,2,3]} &= \texttt{1 + 2 + 3 + 0} \\
\texttt{mySum []} &= \texttt{0}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{prod [1,2,3]} &= \texttt{1 * 2 * 3 * 1} \\
\texttt{prod []} &= \texttt{1}
\end{aligned}
$$

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

Patterns, Guards, and More

Patterns

- used to match specific cases

Patterns

- used to match specific cases
- defined by

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{def}{=} & \texttt{\_} & \text{wildcard} \\
& | & x & \text{variable pattern} \\
& | & x\texttt{@}\langle pat \rangle & \text{``as'' pattern} \\
& | & \texttt{[}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\
& | & \texttt{(}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& | & C \ \langle pat \rangle \ \ldots \ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

Patterns

- used to match specific cases
- defined by

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{def}{=} & \texttt{\_} & \text{wildcard} \\
& | & x & \text{variable pattern} \\
& | & x\texttt{@}\langle pat \rangle & \text{``as'' pattern} \\
& | & \texttt{[}\langle pat \rangle \texttt{,} \ldots \texttt{,} \langle pat \rangle\texttt{]} & \text{list pattern} \\
& | & \texttt{(}\langle pat \rangle \texttt{,} \ldots \texttt{,} \langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& | & C\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

- _ matches everything and ignores the result

Patterns

- used to match specific cases
- defined by

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{def}{=} & \texttt{\_} & \text{wildcard} \\
& \mid & x & \text{variable pattern} \\
& \mid & x\texttt{@}\langle pat \rangle & \text{``as'' pattern} \\
& \mid & \texttt{[}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\
& \mid & \texttt{(}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& \mid & C\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

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

Patterns

- used to match specific cases
- defined by

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{def}{=} & \texttt{\_} & \text{wildcard} \\
& | & x & \text{variable pattern} \\
& | & x\texttt{@}\langle pat \rangle & \text{“as” pattern} \\
& | & \texttt{[}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\
& | & \texttt{(}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& | & C\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

- `_` 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

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{def}{=} & \texttt{\_} & \text{wildcard} \\
& \mid & x & \text{variable pattern} \\
& \mid & x\texttt{@}\langle pat \rangle & \text{``as'' pattern} \\
& \mid & \texttt{[}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\
& \mid & \texttt{(}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& \mid & C\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

- `_` 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 type constructor (example type constructors: `(:)` and `[]` for lists, `True` and `False` for Boolean values, ...)

Patterns

- used to match specific cases
- defined by

$$
\begin{array}{rcll}
\langle pat \rangle & \stackrel{\mathit{def}}{=} & \texttt{\_} & \text{wildcard} \\
& \mid & x & \text{variable pattern} \\
& \mid & x\texttt{@}\langle pat \rangle & \text{“as” pattern} \\
& \mid & \texttt{[}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\
& \mid & \texttt{(}\langle pat \rangle\texttt{,} \ldots \texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\
& \mid & C\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern}
\end{array}
$$

- `_` 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 type constructor (example type 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

$$
\begin{array}{llll}
\texttt{case}\ e\ \texttt{of} & \langle pat_1 \rangle & \texttt{->} & e_1 \\
 & \vdots & & \\
 & \langle pat_n \rangle & \texttt{->} & e_n
\end{array}
$$

- 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

$$\begin{array}{llll} \texttt{case}\ e\ \texttt{of} & \langle pat_1 \rangle & \texttt{->} & e_1 \\ & \vdots & & \\ & \langle pat_n \rangle & \texttt{->} & e_n \end{array}$$

- 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

Pattern Matching Examples

```
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
- $\langle pat \rangle$ | b

Pattern Guards

- a pattern may be followed by a guard *b*
- $\langle pat \rangle$ `|` *b*
- where *b* is a Boolean expression

Pattern Guards

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

Examples

```
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 a function

Definition

a function is of higher-order if

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

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 a function

Examples

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

Sections

- abbreviation for partially applied infix operators
- `(x `op`)` abbreviates `(\y -> x `op` y)`
- `(`op` y)` abbreviates `(\x -> x `op` y)`

Definition

a function is of higher-order if

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

Examples

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

Sections

- abbreviation for partially applied infix operators
- `(x `op`)` abbreviates `(\y -> x `op` y)`
- `(`op` y)` abbreviates `(\x -> x `op` y)`

Examples

```
ghci> twice (*2) 10
40
```

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

```haskell
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 `foldr f b` $[x_1, x_2, \ldots, x_n]$

$$
\begin{array}{llllllll}
= & \texttt{foldr f b} & (x_1 & : & (x_2 & : & \cdots & (x_n & : & \texttt{[]})\cdots)) \\
= & & (x_1 & \texttt{`f`} & (x_2 & \texttt{`f`} & \ldots & (x_n & \texttt{`f`} & b\ )\ldots))
\end{array}
$$

"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` $[x_1, x_2, \ldots, x_n]$

$$
\begin{array}{lllllll}
= & \texttt{foldr f b} & (x_1 \;\texttt{:} & (x_2 \;\texttt{:} & \cdots & (x_n \;\texttt{:} & \texttt{[]}\,)\cdots)) \\
= & & (x_1 \;\texttt{`f`} & (x_2 \;\texttt{`f`} & \ldots & (x_n \;\texttt{`f`} & b\,)\ldots))
\end{array}
$$

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` $[x_1, x_2, \ldots, x_n]$

$$
\begin{array}{lllllll}
= & \texttt{foldr f b} & (x_1 \;\texttt{:}\; & (x_2 \;\texttt{:}\; & \cdots & (x_n \;\texttt{:}\; & \texttt{[]})\cdots)) \\
= & & (x_1 \;\texttt{`f`}\; & (x_2 \;\texttt{`f`}\; & \ldots & (x_n \;\texttt{`f`}\; & b\;)\ldots))
\end{array}
$$

This Pattern is Very General

add dummy argument

- 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`)

Exercises (for October 22nd)

1. read chapters 1 and 2 of Real World Haskell
2. work through lessons 4 to 6 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 `map (+1) [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]`.