3: Functional paradigm

Programming Languages, Technologies and Paradigms

Escuela Técnica Superior de Ingeniería **Informática** etsinf

Summary

Introduction to Functional Programming

PART I: Types in Functional Programming

1. Functional types. Algebraic types.
2. Predefined types.
3. Polymorphism: genericity, overloading and coercion. Inheritance in Haskell.

PART II: Models of computation in functional programming.

4. Operational model.

PART III: Advanced features

5. Anonymous functions and composition.
6. Iterators and compressors (foldl, foldr).

Objectives

- Identifying the foundations of the functional programming paradigm: referential transparency and absence of side effects and global variables. Functions as first-class citizens.
- Understanding algebraic and functional datatypes as used in modern functional languages
- Understanding the relationship between polymorphism and inheritance, as well as their use in functional and object-oriented languages.
- Solving problems using partial functions that may not terminate.
- Understanding and applying currying, partial application and higher-order.
- Understanding the reduction-based computational model of functional programming, in connection with evaluation strategies.
- Applying iteration and compression schemata in problem solving.
- Understanding "mapreduce" and its connection with parallelism. Understanding how to use it in information processing on the web.

Introduction

Looking for solutions to the *Software Crisis*

- New developments in Software Engineering for the analysis and design of big software projects
- Providing appropriate systems for program verification/ testing
- New techniques for program synthesis: can (correct) *executable code* be obtained from a formal *specification*?
- New designs for computer architectures (parallel processing techniques)
- **Alternative to the traditional (imperative) model of computation**

Why functional programming matters?

- Functional languages are increasingly popular in different contexts:
  - **Haskell** (a pure functional language is used in many fields: https://wiki.haskell.org/Haskell_in_industry)
  - **Scala** (functional & OO, used to develop Twitter)
  - **Erlang** (functional & concurrent, essential for Whatsapp or Facebook, see http://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/)

Why functional programming matters?

- Many programming languages are borrowing typical functional programming features:
  - **Python**: higher-order, map, reduce, etc
  - **JavaScript**: higher-order, lambda abstractions, closures, map, etc.
  - **Java**: higher-order, lambda abstractions
  - **Ruby**: higher-order, lambda abstractions, partial application
  - **PHP**: higher-order, lambda abstractions, etc

Distinctive features

- Absence of side effects
- Functions as first-class citizens
- User defined types and datastructures
- Partial application
- Evaluation strategies

No side effects - Functions

- Absence of side effects; functions as first-class citizens
  - Absence of side effects

    The outcome of a function depends on its arguments *only* (referential transparency)
  - Functions as first-class citizens

    Functions can be arguments of other functions (higher-order) or returned as the outcome of a function call (for instance, by means of a partial application).

```
$ map sqr [1,2,3]
[1,4,9]
```

```
$ map (inc 1) [1,2,3]
[2,3,4]
```

where:

```
inc:: Int -> (Int -> Int)
inc x = (x +)
```

Partial application of functions

- Every function

$$f : D_1 \times D_2 \times \ldots \times D_k \to E$$

can be presented in *curried* version as follows

$$f' : D_1 \to (D_2 \to (\ldots \to (D_k \to E) \ldots)$$

where each value in D_1 is given a function of (k-1) arguments (and so on and so forth)

- **Currying** enables the **partial application** of functions to their arguments.

Partial application of functions

- In the partial application of a curried function the number of *passed* parameters is *smaller* than the number of *formal* parameters in its definition
- Example: arithmetic operators

```
(+)  :: Int -> Int -> Int
$ (2 +) 5
7
```

- Example: functions defined as partial applications

```
add_2 :: (Int -> Int)
add_2  = (2 +)
$ add_2 5
7
```

Evaluation strategies

- Evaluation strategies

```
three :: Int → Int
three x  = 3

infinite :: Int
infinite = infinite +1
```

```
     three infinite
=  {definition of infinite}
     three (infinite +1)
=  {definition of infinite}
     tres ((infinite +1)+1)
=  {...}
```

```
     three infinite
=  {definition of three}
     3
```

Lazy evaluación: arguments are evaluated only if *necessary*; termination is guaranteed (when possible).

Eager evaluation: arguments in a function call are *all* evaluated before calling the function.

Summary

Introduction to Functional Programming

PART I: Types in Functional Programming

1. Functional types. Algebraic types.
2. Predefined types.
3. Polymorphism: genericity, overloading and coercion. Inheritance in Haskell.

PART II: Models of computation in functional programming.

4. Operational model.

PART III: Advanced features

5. Anonymous functions and composition.
6. Iterators and compressors (foldl, foldr).

Functional types

- The type constructor -> builds a functional type out from two given types.

  Example: type MyType = (Int -> Int)

  fib :: MyType

- In general a_1 -> a_2 -> … -> a_n is a functional type whose values are those functions having this type
  - For instance, function not is a value of type Bool -> Bool
  - The type of function (2 +) is Int -> Int
  - The type of function map is (a -> b) -> [a] -> [b]

what is the type of map (2 +)?

Functional types

- The operator -> is right associative

    a -> b -> c   is equivalent to   a -> (b -> c)

    and different from   (a -> b) -> c

- The functional application operator is left associative

    f a b   is equivalent to (f a) b and different from f (a b)

Example

$ not not false   error

Algebraic types (Examples)

```
Data Status = Single | Married

status :: Status -> String
status Married = ”Brought to the altar”
status Single = ”Free like a bird"

: status Single
”Free like a bird"
```

```
Data Status = Married Bool | Single Int

status :: Estado -> String
status (Married x) = if x then ”He is happy" else ”He is unhappy”
status (Single x) = ”Still  "++(show x)++" years to be married"

:status (Married True)
”He is happy"
```

Algebraic Types (Examples)

```
type Name = String
type Position = String
type Age = Int
type Course = Int

data Person = Student Age Name Course |
              Professor Name Position |
              Director Name

namePerson :: Person -> Name
namePerson  (Student e n c) = n
namePerson  (Professor n c) = n
namePerson  (Director n) = n

: namePerson (Professor "Albert" "Assistant")
"Albert”
```

Algebraic types

- The natural numbers can be defined as follows

```
data Nat = Zero | Suc Nat
```

Recursive declaration

  - Values: Zero, Suc(Suc Zero), Suc(Suc(Suc(Suc(Suc Zero)))),…
  - Arithmetic operators over the naturals: addition and product

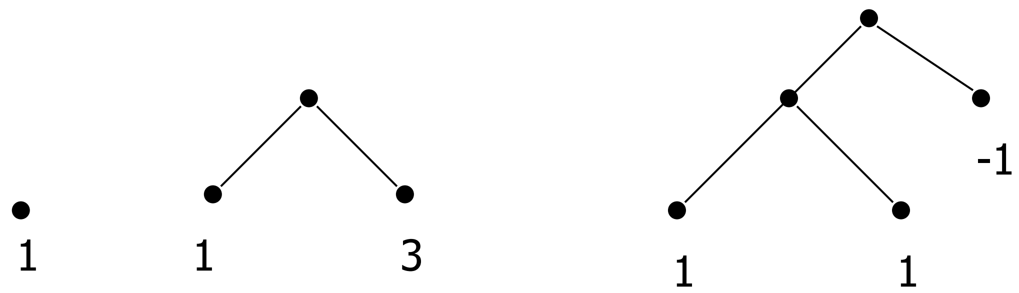
```
(+$) :: Nat -> Nat -> Nat
x +$ Zero = x
x +$ (Suc y) = Suc (x +$ y)

(*$) :: Nat -> Nat -> Nat
x *$ Zero = Zero
x *$ (Suc y) = x +$  (x *$ y)
```

Algebraic types

- Example: the type of binary trees containing elements of type a can be defined as an algebraic type as follows:

  data BinTree a = L a | B (Tree a) (Tree a)



L 1   B (L 1) (L 3)   B (B (L 1) (L 1))(L –1)

Algebraic types

- enumeration

simple { **data** Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab

- types whose values are built by using values of other types

structured { **data** Either = Left Bool | Right Char

Data constructors

in general

parametric { **data** Either a b = Left a | Right b

Type variables

Use of expressions with Left and Right in patterns

either :: (a -> c) -> (b -> c) -> Either a b -> c

either f g (Left x) = f x

either f g (Right y) = g y

predefined

data T a_1 . . . a_n = C_0 t_{01} . . . t_{0k_0} | . . . | C_m t_{m1} ... t_{mk_m}

Type constructor

Algebraic types

- The values of an algebraic data type are expressions containing constructor symbols only
- They are obtained by using the type definition as a grammar, so that:
  - Data constructors => terminal symbols
  - Type constructors => nonterminal symbols
  - E.g. Zero, Suc Zero, Suc (Suc Zero),…
- Patterns are expressions consisting of constructor symbols and variables
  - Patterns represent sets of values. For instance, (Suc n) can be used to represent the set of positive naturals numbers

Algebraic types: pattern matching

- A expression e **matches** a pattern *p* (*pattern matching*) if e is an **instance** of *p* (by giving values to the **variables** in *p*)
- Pattern matching is a standard function definition mechanism in functional programs
- With pattern matching, functions are defined by describing their outcome for the **set of values** given by a **pattern**

Algebraic types: pattern matching

- Some functions defined by *pattern matching*:
  - Exclusive-or:

```
exOr :: Bool -> Bool -> Bool
exOr True y = not y
exOr False y = y
```

  - if _ then _ else

```
cond :: Bool -> a -> a -> a
cond True x y = x
cond False x y = y
```

Summary

Introduction to Functional Programming

PART I: Types in Functional Programming

1. Functional types. Algebraic types.
2. Predefined types.
3. Polymorphism: genericity, overloading and coercion. Inheritance in Haskell.

PART II: Models of computation in functional programming.

4. Operational model.

PART III: Advanced features

5. Anonymous functions and composition.
6. Iterators and compressors (foldl, foldr).

Predefined types (Char)

- Are written as follows:

  'a', 'b', '0', '\n', '\r',...

- Predefined functions for char processing:

  ord :: Char -> Int *from character c we get its integer code ord c*

  chr :: Int -> Char *from an integer we get the corresponding char*

  Comparison operator(s)

  : ord 'b'

  98

  : chr 98

  'b'

  : 'A' < 'a'

  True

  : 'b' == chr 98

  True

Predefined types (Char)

- More functions:
  - isAlpha, isAlphaNum, isDigit, isLower, isUpper :: Char -> Bool
  - toLower, toUpper :: Char -> Char
  - putChar :: Char -> IO ()

Predefined types (tuples)

- Tuples consist of (two or more) components of possibly different types.

  (Int,Char)

  (Char,(Int,Char))

  (Char,Int,Char)

  . . .

- Functions for tuples of 2 components (pairs)
  - fst :: (a,b) -> a

    fst (x,y) = x
  - snd :: (a,b) -> b

    snd (x,y) = y

Exercise

- Define a function that takes two numbers and returns a pair with both numbers in increasing order.

Exercises

- Define a function nextLetter :: Char -> Char that transforms each *letter* in the alphabet into the next one, whilst the other characters remain untouched. Assume that nextLetter 'Z' = 'A' and nextLetter 'z' = 'a'.
- Use a tuple (d,m,a) of natural numbers to represent a date, where d, m and a refer the day, month and year, respectively. Define a function that, given the date of birth of a person and the current date returns its age in years.
- Let sigma and pi be functions given as follows:

  sigma f a b = $\Sigma_{a \leq i \leq b}$ f i

  pi f a b = $\Pi_{a \leq i \leq b}$ f i

  provide executable recursive definitions for both of them, including appropriate type declarations.

Predefined types (String)

- Definition: **type** String = [Char]
  - Char sequences enclosed between double quotes
  - Compared by using the lexicographic ordering

    $ "Juan" < "Juana"
    True

    $ "Palo" < "palo"
    True

  - Values of (some) types can be transformed into strings

    show :: Show a => a -> String

    $ show 6
    "6"

Predefined types (numbers)

- Numeric types: Int, Float
  - Int: bounded range integers
  - Float: simple precision floating point real numbers (e.g., 0.345, -23.12, 231.61e7, 46.7e-2,...)
- Haskell supports more numeric types
  - Integer: unbounded integer numbers
  - Double: double precision floating point real numbers
  - Complex: complex numbers
  - Rational: rational numbers (library Ratio)

Algebraic types (lists)

The predefined type *list* corresponds to a recursive algebraic polymorphic type as follows:

Type variable

```
data [a] = [ ]  |  a : [a]
```

Type constructor

Data constructors

Notation for lists

- Since lists are pervasive in functional programming, several suitable notations have been developed for them: for instance,

  1:2:3:[]   (equivalently 1:(2:(3:[])))

  [1,2,3]

  1:[2,3]

  correspond to the same list

  (:) is right *associative*

- The notation for **arithmetic lists** permits the definition of sequences of values of enumerated types

[2..10]	is	[2,3,4,5,6,7,8,9,10]
[1..]	is	[1,2,3,4,...
[1,3..10]	is	[1,3,5,7,9]
['a'..'e']	is	"abcde"

List comprehension

The notation for **list comprehension** borrows the usual notation for set-theoretic expressions

<u>Example</u>. The set of squares of odd integers between 1 and 5

$$\{x*x \mid 1<=x<=5\ ,\ \mathrm{odd}\ x\}$$

$$[\ x*x \mid x<-\underbrace{[1..5]}_{\text{arithmetic list}},\ \mathrm{odd}\ x\ ]$$

List comprehension

The notation for **list comprehension** borrows the usual notation for set-theoretic expressions.

$$[\ x*x \mid \underbrace{\overbrace{x{<}{-}[1..5]}^{\text{generator}},\ \overbrace{\text{odd } x}^{\text{guard}}}_{\text{qualifier}}\ ]$$

List comprehension

Syntax of list comprehension :

[e | Q]

expression (whose variables take value in Q)

qualifier: a (possibly empty) sequence of ***generators*** and ***guards***

p <- xs

g

pattern of type a

exp. of type [a]

boolean expression

List comprehension

- Semantics of list comprehension:

  $[\ e\ |\ p_1\text{<-}xs_1,\ g_1,\ldots,\ p_n\text{<-}xs_n,\ g_n\ ]$

  - the generators are used from left to right, where the rightmost one is first changed when necessary
  - the guards are evaluated from left to right
  - the returned list collects the values which are obtained when e is evaluated with all variables instantiated by the generators provided that all guards are satisfied on them

```
$ [(x,y) | x<-[1..5], odd x, y<-[x..5], odd y]
[(1,1), (1,3), (1,5), (3,3), (3,5), (5,5)]
$ [(x,y) | (x:y:xs) <- ["abcde", "f", "ghi"]]
[('a','b'),('g,'h')]
```


Exercise: Use list comprehension to define functions sigma and pi

Examples

- map f xs = [f x | x <- xs]

- filter p xs = [x | x <- xs, p x]

- repetitions y xs = length [() | x <- xs, y == x]

- divisors n = [i | i <- [1..n], n `mod` i == 0]

- isMember y xs = not (null [() | x<-xs, y == x])

Exercises

- Define a function elimDups :: [Int] -> [Int] that removes from the given list all duplicate elements that are contiguous. For instance

  elimDups [1,2,2,3,3,3,1,1]=[1,2,3,1]

- Define functions

  any, all :: (a -> Bool) -> [a] -> Bool

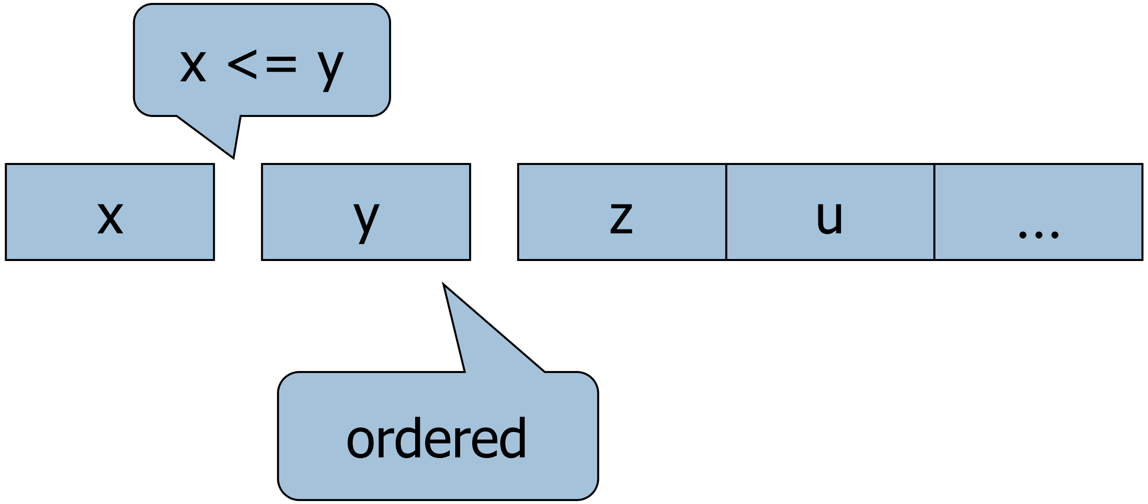
  satisfying the following specifications

  any p xs $\Leftrightarrow \exists\, x \in xs : p\ x$

  all   p xs $\Leftrightarrow \forall\, x \in xs : p\ x$

Exercises

- Define a function that checks whether a list of integers is ordered.



- Define a function that builds a list of n copies of x (Hint: use list comprehension).

Operations on lists

- Properties of a list
  - Length of a list: length :: [a] -> Int

    length [ ] = 0

    length (x:xs) = 1 + length xs

  - Empty list?: null :: [a] -> Bool

    null [ ] = True

    null (x:xs) = False

Operations on lists

- Combination of lists
  - Concatenation: (++) :: [a] -> [a] -> [a]

    [ ] ++ xs = xs

    (x:xs) ++ ys = x:(xs ++ ys)
  - Concatenation with flattening: concat :: [[a]] -> [a]

    concat [ ] = [ ]

    concat (xs:xss) = xs ++ concat xss
  - Combination: zip :: [a] -> [b] -> [(a,b)]

    zip [ ] xs = [ ]

    zip (x:xs) [ ] = [ ]

    zip (x:xs) (y:ys) = (x,y) : zip xs ys

  Example $ zip [1, 2, 3] [”a”, ”b”, ”c”]

  [(1,”a”), (2,”b”), (3,”c”)]

  where zip :: [Int] -> [String] -> [(Int, String)]

Operations on lists

- Componentwise access to a list
  - Head of a list: head :: [a] -> a

    head (x:xs) = x
  - Last element of a list: last :: [a] -> a

    last [x] = x

    last (x:xs) = last xs
  - Indexed access: (!!) :: [a] -> Int -> a

    (x:xs) !! 0 = x

    (x:xs) !! n = xs !! (n-1)
- Sublists of a list
  - Beginning of a list: init :: [a] -> [a] -- all but the last element
  - Tail of a list: tail :: [a] -> [a] -- all but the first element

Exercise 1

- Define a function position returning the position of an element within a list.

  position :: a -> [a] -> Int

  $ position "b" ["a", "b", "c"]

  2

Hint: mark each element of the list with its position and search the one for the desired element

["a", "b", "c"] → [("a",1), ("b",2), ("c",3)]

zip

Exercise 1

zip [”a”, ”b”, ”c”] [1, 2, 3] = [(”a”,1), (”b”,2), (”c”,3)]

- We know that the list xs has (length xs) elements

zip xs [1.. length xs]

Exercise 1

Return the (first) position where the desired element occurs

```
position x xs = [pos | (x',pos) <- zip xs [1..], x' == x]
```

```
position "b" ["a", "b", "c"] = [2]
```

We obtain a list rather than a number

Exercise 1

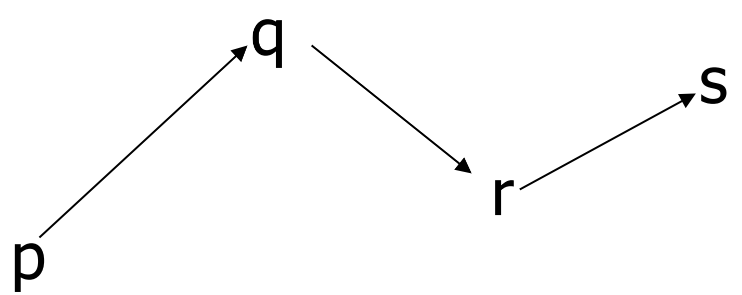
Solution:

```
position :: a -> [a] -> Int
position x xs = head [pos | (x',pos) <- zip xs [1..], x' == x]
```

the first element of the list is returned

Exercise 2

- Define a function that computes the length of a path.



- Representation:

  type Point = (Float,Float)

  type Path = [Point]

  examplePath = [p,q,r,s]

  pathLength = distance p q + distance q r + distance r s

Exercise 2

- Two useful functions:

  init [p, q, r, s] = [p, q, r]

  tail [p, q, r, s] = [q, r, s]

- Combination of both lists: zip

  zip … = [(p,q), (q,r), (r,s)]

Exercise 2

Solution:

```
pathLength :: Path -> Float
pathLength xs = sum' [distance p q | (p,q) <- zip (init xs) (tail xs)]


sum' :: [Float] -> Float
sum' [] = 0
sum' (x:xs) = x + sum' xs


distance :: Point -> Point -> Float
distance (p1,p2) (q1,q2) = sqrt (sqr (p1 - q1) + sqr (p2 - q2))
```

Operations on lists

- Ordering a list
  - By insertion in an ordered list

```
insert x [ ] = [x]
insert x (y:ys)
  | x<=y = (x:y:ys)                    -- [x]++(y:ys)
  | otherwise = y : (insert x ys)

inorder [ ] = [ ]
inorder (x:xs) = insert x (inorder xs)
```

Operations on lists

- More efficient ordering functions: mergeSort
  - divide a list into two halves
  - order each of the two halves
  - Put together the two ordered halves

Operations on lists

```
mergeSort xs = merge (mergeSort front) (mergeSort back)
        where size = length xs `div` 2
              front = take size xs
              back = drop size xs
```

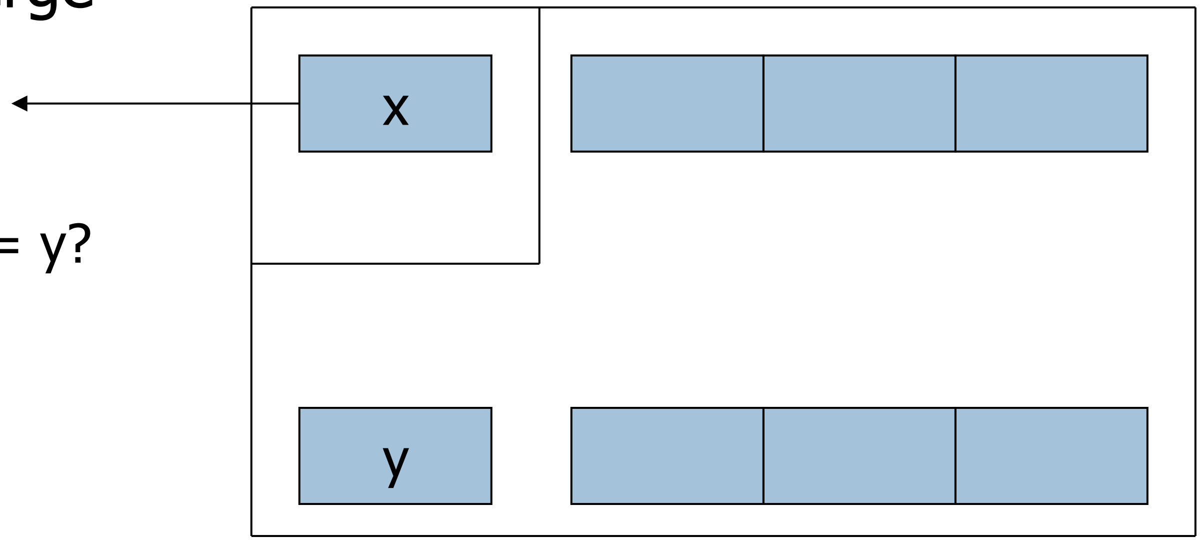
Only works if *both* front and back are *smaller* than xs

Operations on lists

```
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs | size > 0 =
   merge (mergeSort front) (mergeSort back)
   where
      size = length xs `div` 2
         front = take size xs
         back = drop size xs
```

Operations on lists

merge



x <= y?

merge [1, 3] [2, 4] → 1 : merge [3] [2, 4]

→ 1 : 2 : merge [3] [4]

→ 1 : 2 : 3 : merge [] [4]

→ 1 : 2 : 3 : [4] → [1,2,3,4]

Operations on lists

Solution:

alias of a pattern
(*as-pattern*)

```
merge :: [Int] -> [Int] -> [Int]

merge a@(x:xs)  b@(y:ys)

   | x <= y= x : merge xs b

   | otherwise      = y : merge a   ys

merge [ ] ys        = ys

merge xs [ ]        = xs
```

Operations on lists

- Transformation of lists
  - Reverse: reverse :: [a] -> [a]

    reverse [ ] = [ ]

    reverse (x:xs) = reverse xs ++ [x]
  - Application of a function to the components of a list:

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

    map f [ ] = [ ]

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

elements transformed by f

Summary

Introduction to Functional Programming

PART I: Types in Functional Programming

1. Functional types. Algebraic types.
2. Predefined types.
3. Polymorphism: genericity, overloading and coercion. Inheritance in Haskell.

PART II: Models of computation in functional programming.

4. Operational model.

PART III: Advanced features

5. Anonymous functions and composition.
6. Iterators and compressors (foldl, foldr).

Coercion

- Coercion is explicit in Haskell. There are functions that transform some types into others.

Examples:

- Numeric conversion is not automatic. There are specific functions for that:

```
import GHC.Float
:t float2Double                          :t double2Float
float2Double:: Float -> Double           double2Float:: Double-> Float
```

- Function *show* transforms any predefined type into a string. It can be used to coerce integers into strings

show 3 =“3”

Genericity

- A function is generic if it has a polymorphic type (i.e., contains type variables).

Example:

```
either :: (a -> c) -> (b -> c) -> Either a b  -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

Overloading

- Overloading is implemented in Haskell through the notion of **type classes.**
  - Type classes enable the use of parametrization for defining overloaded functions for some types which must be included into a type class.

Example: + :: Num a ⇒ a -> a -> a

restricts the use of the addition operator to those types belonging to the class Num.

  - The **type class** declaration specifies the operations that can be used with any type which has been previously included in (i.e., made an instance of) the class.

Example: **class** Eq a **where**

(==), (/=) :: a -> a -> Bool

  - The type of overloaded functions includes a reference to the class where the usable types are supposed to be included in advance.

Example $:t (==)

== :: Eq a => a->a->Bool

Overloading

- Each **instance** of a class is obtained by providing an specific implementation of the class operations for the targetted type.

Example: data Nat = Cero | Suc Nat

```
instance Eq Nat where
Cero == Cero = True
Suc x == Suc y = x == y
_ == _ = False
```

- Algebraic types can be added to a type class by using a **deriving** clause in the type definition.

Example: **data** Bool= False | True **deriving** (Eq, Ord, Enum)

the operations are given an implementation on the basis of the syntactic structure of the type definition (for instance, False < True).

Some predefined classes

- Eq((==), (/=))

includes all predefined types except IO, (->)

- Ord((<), (<=), (>=), (>), max, min)

includes all predefined types except IO, IOError, (->)

- Num((+), (-), (*), negate, abs, signum, …)

includes all numeric types (Int, Integer, Float, Double, Ratio)

- Show(show,…)

includes all predefined types except IO, (->)

Class inheritance

- Inheritance can be used to define some classes

Example: Ord is a subclass of Eq that provides a default implementation of <=, >=, > as follows

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>) :: a -> a -> Bool
   x <= y       =  (x<y) || (x==y)
   x >= y       =  (x>y) || (x==y)
   x >  y       =  not (x <= y)
```

Extending classes

- Instances of extended classes can be defined using **instance**.

Example:

```
instance (Eq Nat) => Ord Nat where
  Cero < Suc x = True
  Suc x < Suc y = x < y
  _ < _ = False
```

Note: the context (Eq Nat) => is not necessary because Ord extends Eq, but Eq should be instantiated with Nat before instantiating Ord with Nat

Class instances

- Dealing with generic types, we may need contexts (witnessing class membership) for their arguments.
- Example:

```
data Figure = Circle Float | Rect Float Float deriving (Eq, Ord, Show)
```

Float must be an instance of Eq, Ord, Show (and it is predefined in that way)

```
data Tree a = Void | Branch a (Tree a) (Tree a)
instance (Eq a) => Eq (Tree a)
Void                == Void                = True
(Branch x l1 r1) == (Branch y l2 r2) = (x==y) && (l1 == l2) && (r1==r2)
_                   == _                   = False
```

a must be an instance of Eq so that == (as used in the second equation) is overloaded for values of type a.

Exercises

Consider the algebraic type Nat as defined above:

1. Overload arithmetic operators (+ and *) so that you can use them to add and multiply values from Nat
2. Overload **show** from class Show so that values from Nat are displayed with the usual numeric shape: Zero as 0, Suc Cero as 1, …
3. Overload the necessary methods from Enum so that we can define arithmetic lists with values of type Nat. For instance, [Zero..Suc (Suc Zero)] yields [Zero, Suc Zero, Suc (Suc Zero)]
4. Overload operator < from Ord so that we can compare values of type Figure according to the area of the corresponding figure
5. Overload show from class Show so that circles are displayed with their radius enclosed by parentheses (e.g., (2.5)) and rectangles with their sides enclosed by square brackets and separated by commas (e.g., [1.5,2.5])