Advanced Programming II

Lesson 2: Functional Programming in Scala

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Contents

- What is functional programming?
  - Pure and Impure Functions
  - Referential transparency
- Types, values, and immutable objects
- Functions, Recursion, and Tail Recursion
- Deconstructing Data and Patterns
- Immutable lists and functions on lists

Functional programming

- In Informatics, **Functional programming** is a paradigm of declarative programming based on the use of true Mathematical functions.

  (https://en.wikipedia.org/wiki/Functional_programming)

- A function takes a sequence of arguments, performs a calculation with them, and returns a result.
- The value of the same expression is always the same.
- By combining functions, you can construct expressions whose evaluation corresponds to solving a problem.
- There are many functional programming languages (Scala, Haskell, ML, Erlang, etc.).
- And many others that have incorporated functional programming concepts (Java, C++, Python, etc.).

What is functional programming?

- From M. Odersky, in the preface of [Pilquist et al. 2023]:
  - "For me, it's simply an alias for *programming with functions*—that is, a programming style that puts the focus on the functions in a program."
- The key point is what we mean by functions:
  - "pure functional programming restricts functions to be as they are in mathematics: binary relations that map arguments to results"

Pure Functions

- Pure functions are functions that have no side effects.
- What are side effects? E.g.:
  - Modifying a variable.
  - Modify a data structure.
  - Modifying an object´s attribute.
  - Throwing an exception or stopping the program with an error.
  - Printing on console or reading input from the user.
  - Reading or writing on a file.
  - Drawing on the screen.

Pure and Impure Functions

- Although it is possible to write any program using pure functions only, Scala supports both pure and impure functions.
- Side effects, such as mutations, I/O, or the use of exceptions can be useful in many situations.
- Using pure functions has many advantages:
  - modularity,
  - easier to test, reuse, generalize, and reason about,
  - and therefore: less prone to errors.

Benefits of Functional Programming

```
class Bar:
  def buyCoffee(cc: CreditCard): Coffee =
    val cup = Coffee()
    cc.charge(cup.price)
    cup

class CreditCard:
  def charge(price: Double): Unit =
    println("charging " + price)

class Coffee:
  val price: Double = 2.0

val cc = CreditCard()
val bar = Bar()
val cup = bar.buyCoffee(cc)
```

Side effect

- Code is hard to test!
- A credit card shouldn't have to know how to make the charge, or keep records of them.

From [Pilquist et al. 2023].

Benefits of Functional Programming

```
class Bar:
  def buyCoffee(cc: CreditCard, p: Payments): Coffee =
    val cup = Coffee()
    p.charge(cc, cup.price)
    cup

class CreditCard

trait Payments:
  def charge(cc: CreditCard, price: Double): Unit

class SimulatedPayments extends Payments:
  def charge(cc: CreditCard, price: Double): Unit =
    println("charging " + price + " to " + cc)

class Coffee:
  val price: Double = 2.0

val cc = CreditCard()
val p = Payments()
val bar = Bar()
val cup = bar.buyCoffee(cc, p)
```

Side effect

- Although the code is easier to test (using a *stub* that implements Payments correctly) is still difficult to reuse. E.g.., how to implement a method to buy several coffees?

Benefits of Functional Programming

```
class Bar:
  def buyCoffee(cc: CreditCard): (Coffee, Charge) =
    val cup = Coffee()
    (cup, Charge(cc, cup.price))

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) =
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
    val (coffees, charges) = purchases.unzip
    val reduced = charges.reduce((c1, c2) => c1.combine(c2))
    (coffees, reduced)

case class Charge(cc: CreditCard, amount: Double):
  def combine(other: Charge): Charge =
    if cc == other.cc then
      Charge(cc, amount + other.amount)
    else
      throw Exception("Can't combine charges with different cards")
```

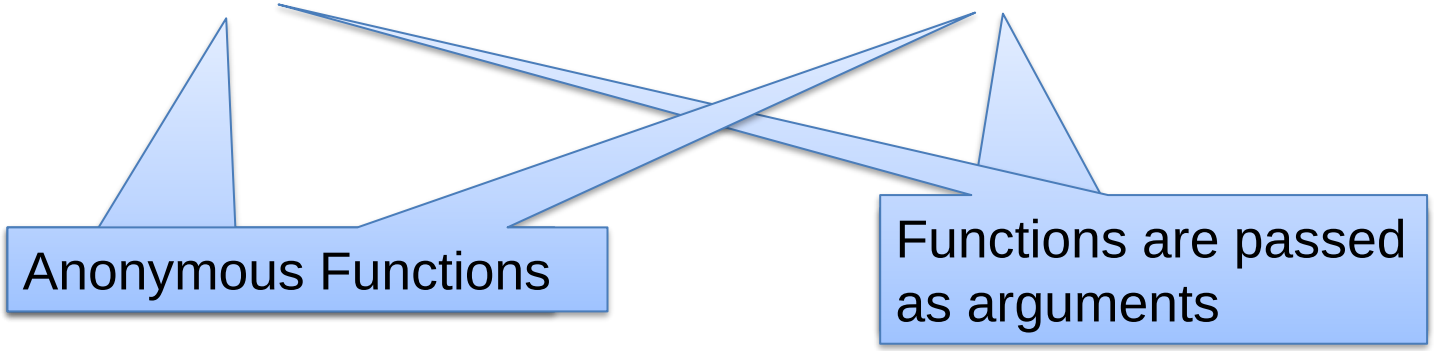
Free of side effects: returns the charge, which will be processed elsewhere.

This solution not only makes easier reusing, both functions are easier to test!

Benefits of Functional Programming

Exercise: What is performed by the function coalesce?

```
def coalesce(charges: List[Charge]): List[Charge] =
  charges.groupBy(_.cc).values.map(_.reduce(_.combine(_))).toList
```



This function is fully reusable and testable.

We will learn to understand and write functions like this throughout the course.

Pure Functions

- A pure function is a function that has no side effects: easier to understand, test and reuse.
- A function f of type A => B is a calculus that relates each of the values in A to exactly one value in B.
  - The result of f(a) is independent of the state of the program or any external process.
- Examples:
  - intToString : Int => String
  - + : Int, Int => Int
  - length : String => Int

Referential transparency

- An expression is said to have referential transparency(or is referentially transparent) if such an expression can be replaced by the result its evaluation without a change in the meaning of the program.
  - E.g.., the expression 2 + 3 can be replaced by 5.
- A function f is pure if the expression f(x) is referentially transparent for every referentially transparent x.
- E.g.: String is immutable, whereas StringBuilder is mutable

```
scala> val s = new StringBuilder("Hello")
val s: StringBuilder = Hello
scala> s.append(", world!").toString
val res3: String = Hello, world!
scala> s.append(", world!").toString
val res4: String = Hello, world!, world!
```

```
scala> val s = "Hello"
val s: String = Hello
scala> s.concat(", World!")
val res9: String = Hello, World!
scala> s.concat(", World!")
val res10: String = Hello, World!
```

Referential transparency

- Is buyCoffee referentially transparent?

```
def buyCoffee(cc: CreditCard): Coffee =
  val cup = Coffee()
  cc.charge(cup.price)
  cup
```

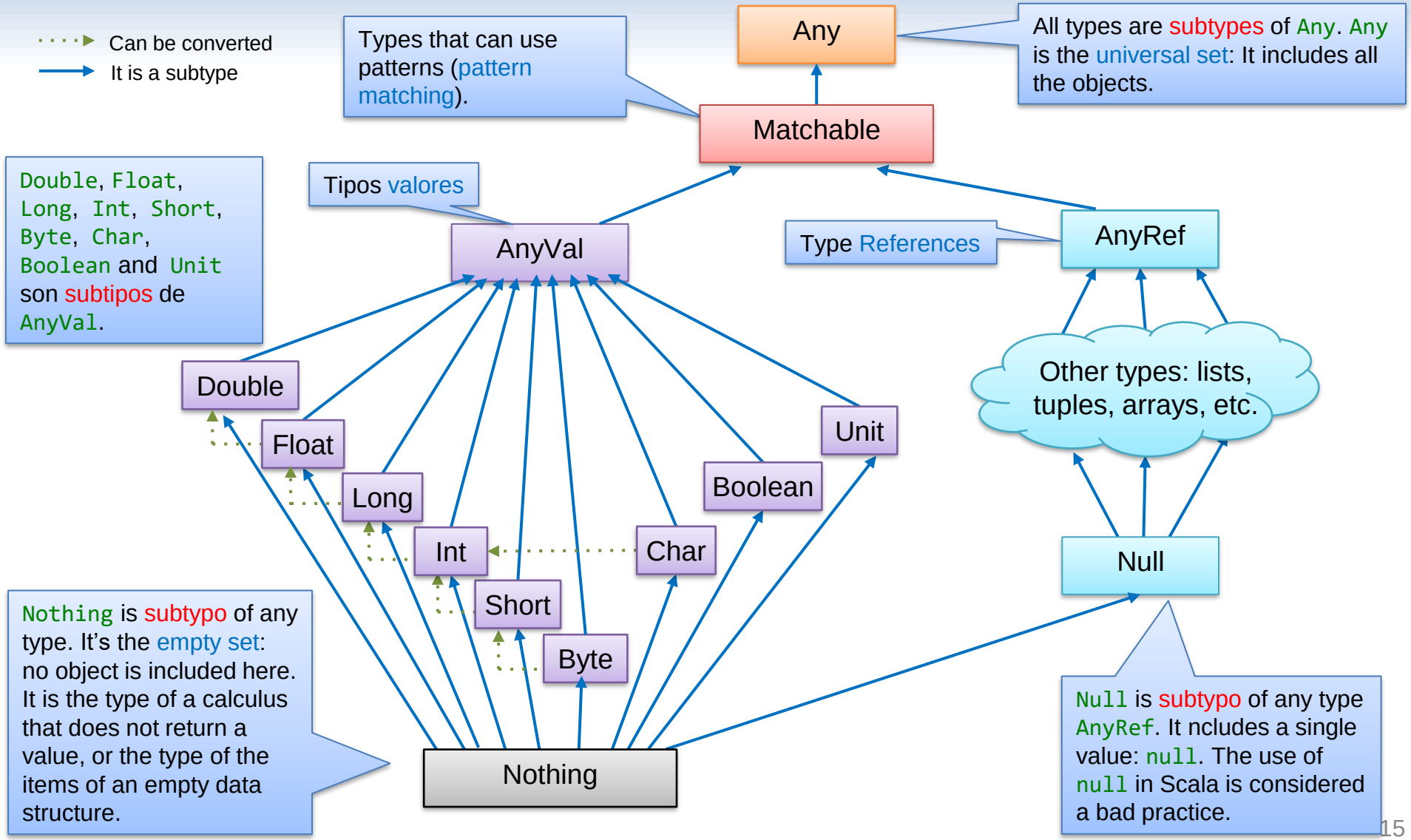
- Is Coffee() equivalent to buyCoffee(aCreditCard)?

- Referential transparency leads us to the substitution model: When expressions are referentially transparent, we can think of computation as substituting each expression by the result of its evaluation.

Types

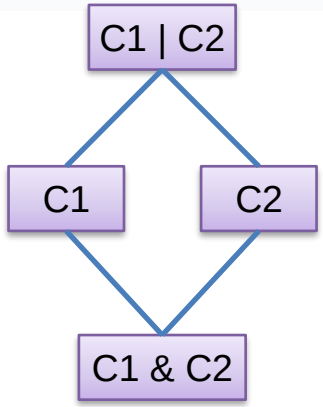
- A type is a set of values with similar features (integers, characters, booleans, lists,, etc.).
- A typed language assigns a type to each value used in a program, as well as to the different operations that manages these values.
- Types allow detecting possible errors when managing values in a program.
- There are two types of typed languages:
  - Dynamic (as Python). Type errors are detected when the program is executed (if any) and never before.
  - Static (as Scala). Type errors are detected when writing the program (compilation time) and before it is executed. In the code written in a static language, in addition to instructions, there are usually type annotations (declarations).

Jerarquía de tipos de Scala



Types **union** and **intersection**

- Given any two types C1 and C2,
  - We can build its intersection type with C1 & C2, and
  - its unión type with C1 | C2



- C1 & C2 represents the closest common subtype to C1 and C2.
- C1 | C2 represents the closest common supertype to C1 and C2.

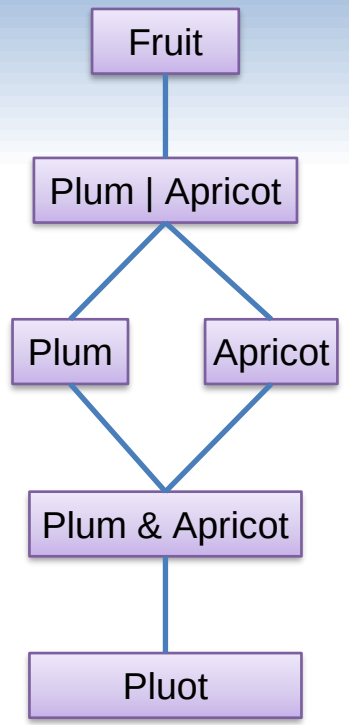
- An intersection type is a subtype (resp. supertype) of all the combinations of the types that constitute them. I.e.:
  - C1 & C2 = C2 & C1,
  - C1 & C2 & C3 < C1 & C2 < C1 < C1 | C3,
  - C1 | C3 < C3 | C1 | C2, ...
- Union types are key in the specification and implementation of Scala's type inference and checking system.

Intersection Types: Example

```
abstract class IntQueue:
  def get(): Int
  def put(x: Int): Unit

import scala.collection.mutable.ArrayBuffer
class BasicIntQueue extends IntQueue:
  private val buf = ArrayBuffer.empty[Int]
  def get() = buf.remove(0)
  def put(x: Int) = buf += x

trait Incrementing extends IntQueue:
  abstract override def put(x: Int) = super.put(x + 1)

trait Filtering extends IntQueue:
  abstract override def put(x: Int) =
    if x >= 0 then super.put(x)

scala> val q = new BasicIntQueue with Incrementing with Filtering
val q: BasicIntQueue & Incrementing & Filtering = anon$...
```

Union types: Example

```
trait Fruit
trait Plum extends Fruit
trait Apricot extends Fruit
trait Pluot extends Plum, Apricot

scala> val plumOrApricot: Plum | Apricot = new Plum {}
val plumOrApricot: Plum | Apricot = anon$1@4e45fbd0
scala> val fruit: Fruit = plumOrApricot
val fruit: Fruit = anon$1@4e45fbd0
scala> val doesNotCompile: Plum | Apricot = fruit
1 |val doesNotCompile: Plum | Apricot = fruit
  |                                     ^^^^^
  |                                     Found:    (fruit : Fruit)
  |                                     Required: Plum | Apricot
```



Type Values

They are predefined and all are subtypes of `AnyVal`.

Operatiing on these types can produce overflow/underflow errors.

- Byte: (1 byte) an integer numerical value between -128 and 127
- Short: (2 bytes) an integer numerical value between -32768 and 32767
- Int: (4 bytes) an integer numerical value between -2147483648 and 2147483647
- Long: (8 bytes) an integer numerical value between -9223372036854775808 and 9223372036854775807
- Float: (4 bytes) Single-precision floating number between -3.40282347×10^{38} and 3.40282347×10^{38}
- Double: (8 bytes) Double-precision floating number between $-1.7976931348623157\times10^{308}$ and $1.7976931348623157\times10^{308}$
- Char: (2 bytes) A single character
- Boolean: (1 byte) A logical value
- Unit: It admits only the value ()

Even if we stay in their ranges, the calculations are not always accurate for types `Double` and `Float`.

Literals

- Integers:
  - `1234`
  - `-987`
  - `121233244L` (L suffix indicates `Long` type)
- Floating point:
  - `1234.5`
  - `1234.5E7` (notación científica, representa 1234.5 x 10^7)
  - `1234.5F` (F suffix indicates `Float` type)
- Characters:
  - `'a'`
  - `'A'`
  - `65` (The character with code65)
  - `'\n'` (A line break)
- Booleans:
  - `true`
  - `false`

Immutable Objects

- An object is inmutable if it is not possible to change its attributes (its internal information remains constant).
- All objects which are subtype of `AnyVal` (`Int`, `Char`, `Double`, `Boolean`, etc.) are immutables.
- Some objects which are subtype of `AnyRef` (`String`, `List`, `Option`, tuples, etc.) are immutables.
- For example:
  - The integer object `8`.
  - The boolean object `false`.
  - The String object `"home"`.
  - The tuple object `(1, true)`.
  - The object `List(1,2,3)`.

Declaration of values

- A **`val`** allows storing a value. Stored value cannot be changed after its initial assignment.

- Examples:

```
val x: Int = 100
val y = 20
val z = 3 + 8
```

In this program, the type of `x is Int` and the vaulue of x is `100` forever.

On most situations the declaration of the type can be omitted (the compiler infers it).

When a value is declared, the expression behind the equal is immediately evaluated (right hand side - RHS) and the result is stored in the identifier.

Integers and simple functions

```
val value1: Int = 10

val value2 = 2

val sum = value1 + value2
val diff = value1 - value2
val product = value1 * value2
val division = value1 / value2
val remaining = value1 % value2

def isEven(x: Int): Boolean =
  x % 2 == 0

def isOdd(x: Int): Boolean =
  !esPar(x)

def threeDifferent(x: Int, y: Int, z: Int): Boolean =
  x != y && x != z && y != z

def between(x: Int, y: Int)(z: Int): Boolean =
  x <= z && z <= y

val checking = between(1, 10)(5)
```

value1 is an Int and contains 10.

value2 is inferred as an Int and contains 2.

Usual operators on integers.

Function isEven is of type Int => Boolean.

Function threeDifferent is of type (Int, Int, Int) => Boolean.

Function between takes two groups of arguments: (Int,Int) => Int => Boolean.

Usage example: ¿is 5 in the range [1,10]?

Recursion

```
def factorial(x: Int): Int =
  if x < 0 then sys.error("Negative number!")
  else if x == 0 then 1
  else x * factorial(x - 1)
```

It is important to define the termination of recursive algorithms. To do this,:

- There must be at least one non-recursive case (base case).
- Recursive cases must lead the algorithm to the base case after a finite number of steps.

```
 factorial(3)
→
 3 * factorial(3 - 1)
→
 3 * factorial(2)
→
 3 * (2 * factorial(2 - 1))
→
 3 * (2 * factorial(1))
→
 3 * (2 * (1 * factorial(1 - 1)))
→
 3 * (2 * (1 * factorial(0)))
→
 3 * (2 * (1 * 1))
→ …
 6
```

Expression size grows during evaluation. Calculus may run out of available memory (stack overflow).

Blocks and scopes

```
def f(x: Int): Int =
  val y = x + 1
  val z = y * y
  z + 10

def factorial(x: Int): Int =
  def aux(acc: Int, i: Int): Int =
    if i == 0 then acc
    else aux(acc * i, i - 1)

  require(x >= 0, “Negative number!")
  aux(1, x)
```

The aux function belongs to the block of the `factorial` function and is local to it.

Precondition: The `factorial` function requires its argument to be non-negative. If it is negative, a result will not be returned and an error will occur (`IllegalArgumentException`) showing the message `Negative number!`.

The `factorial` function will return the result of evaluating the expression `aux(1, x)` if x is not negative.

Tail recursion

```
def factorial(x: Int): Int =
  @tailrec
  def aux(acc: Int, i: Int): Int =
    if i == 0 then acc
    else aux(acc * i, i - 1)

  require(x >= 0, “Negative number!")
  aux(1, x)
```

The aux function uses the accumulator parameter technique to compute the factorial using less memory.

```
 factorial(3)
→ because 3 >= 0
  aux(1, 3)
→ because 3 != 0
  aux(1 * 3, 3 - 1)
→ because 1 * 3 is 3
  aux(3, 3 - 1)
→ because 3 - 1 is 2
  aux(3, 2)
→ because 2 != 0
  aux(3 * 2, 2 - 1)
→ because 3 * 2 is 6
  aux(6, 2 - 1)
→ because 2 - 1 is 1
  aux(6, 1)
→ because 1 != 0
  aux(6 * 1, 1 - 1)
→ because 6 * 1 is 6
  aux(6, 1 - 1)
→ because 1 - 1 is 0
  aux(6, 0)
→ because 0 == 0
  6
```

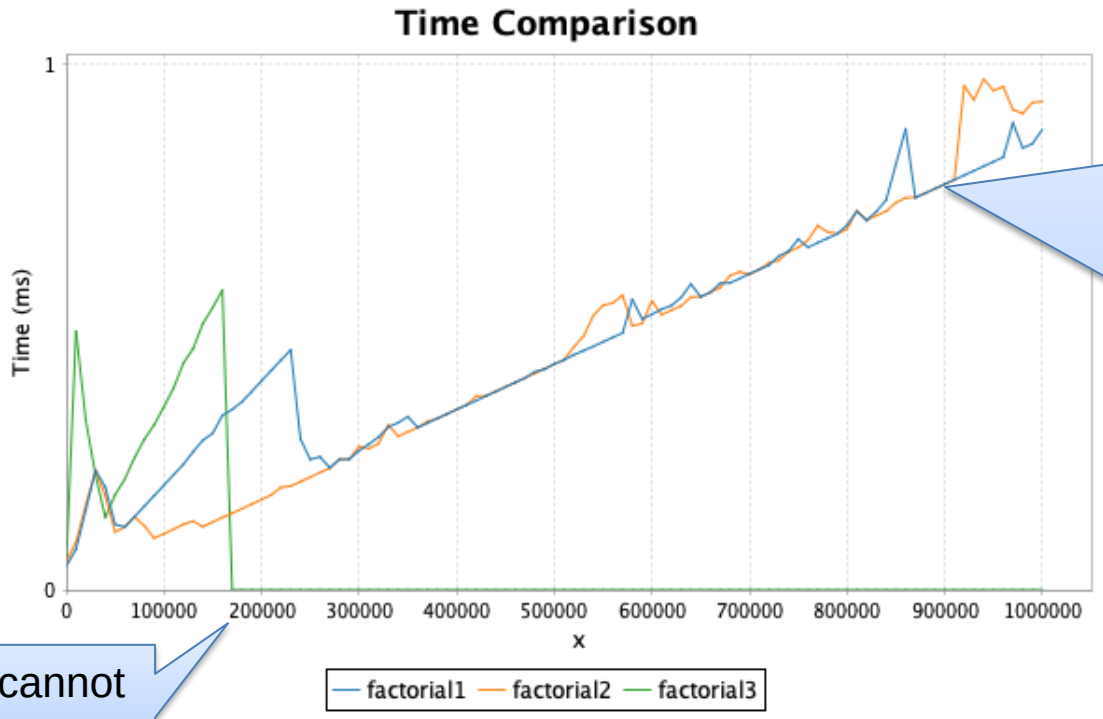
The size of the expressions is limited since it does not grow more than this during the evaluation.

Tail recursion: 3 factorials

```
def factorial1(x: Int): Int =
  @tailrec
  def aux(acc: Int, i: Int): Int =
    if i == 0 then
      acc
    else aux(acc * i, i - 1)
  require(x >= 0, "Negative number!")
  aux(1, x)
```

```
def factorial2(x: Int): Int =
  def aux(acc: Int, i: Int): Int =
    if i == 0 then
      acc
    else aux(acc * i, i - 1)
  require(x >= 0, "Negative number!")
  aux(1, x)
```

```
def factorial3(x: Int): Int =
  require(x >= 0, "Negative number!")
  if x == 0 then
    1
  else x * factorial3(x - 1)
```



Although annotation is needed in some situations, usually the compiler is able to identify the case and optimize it automatically.

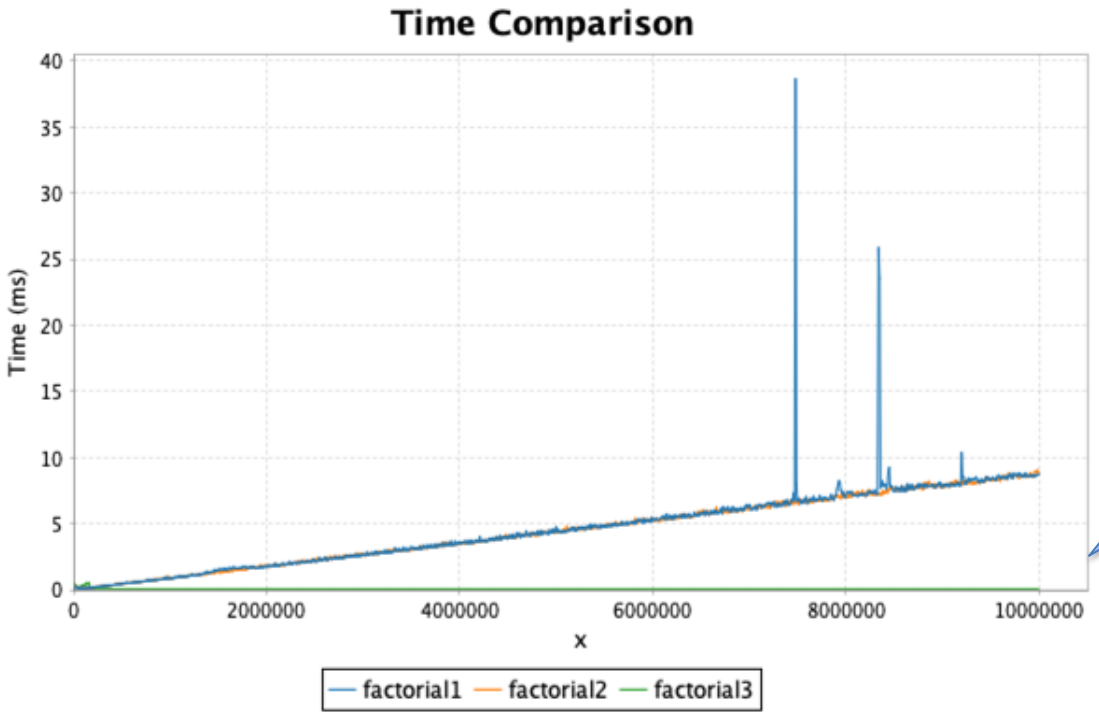
"Normal" recursion cannot handle x values greater than 170000.

Tail recursion: 3 factorials

```
def factorial1(x: Int): Int =
  @tailrec
  def aux(acc: Int, i: Int): Int =
    if i == 0 then
      acc
    else aux(acc * i, i - 1)
  require(x >= 0, “Negative number!")
  aux(1, x)
```

```
def factorial2(x: Int): Int =
  def aux(acc: Int, i: Int): Int =
    if i == 0 then
      acc
    else aux(acc * i, i - 1)
  require(x >= 0, "Negative number!")
  aux(1, x)
```

```
def factorial3(x: Int): Int =
  require(x >= 0, "Negative number!")
  if x == 0 then
    1
  else x * factorial3(x - 1)
```



`factorial1` and `factorial2` work even with extremely large values of x

Tuples

- A tuple is an immutable heterogeneous data structure: it allows grouping several values of different types.

```
val t1: (Int, String) = (10, “Hello")
val t2 = (true, 2, “World")
```

A tuple whose components belong to types `Int` and `String`.

Inferred type: `(Boolean, Int, String)`

```
val first = t2(0)
val second = t2(1)
val third = t2(2)
```

Indexing. The first component has an index 0.

```
val (e11, e12) = t1
val (e21, e22, e23) = t2
```

Deconstructing tuples.

Parametric polymorphism

- Functions that work with arbitrary types adjusted to a type scheme.

`A` is another parameter of `duplicate` but, when invoked, a type must be passed instead of a value.

`duplicate` takes an `x` value of an arbitrary type `A` and returns a tuple with two copies of such a value.

```
def duplicate[A](x: A): (A, A) =
  (x, x)

val result1 = duplicate[Int](10)
val result2 = duplicate("Hello")
```

Returns the tuple `(10, 10)` whose type is `(Int, Int)`.

Returns the tuple (“Hello”, “Hello”) whose type is `(String, String)` (inferred).

Parametric polymorphism

Given an arbitrary type `A`, `twice` it takes a function of `A` in `A` and a value of type `A` and returns a result of type `A`.

```
def twice[A](f: A => A, x: A): A =
    f(f(x))

val result3 = twice((x: Int) => x * x, 2)
val result4 = twice((x: String) => x + x, “Hello")
```

`duplicate2` takes an arbitrary value of type `A` and an arbitrary value of type `B` (possibly different types) and returns a tuple with two copies of each value.

```
def duplicate2[A, B](x: A, y: B): (A, B, A, B) =
  (x, y, x, y)

val result5 = duplicate2(10, “Hello")
```

Type Option

- A value of type `Option[A]` is an immutable object that can be:
  - `None` It does not store any value.
  - `Some(x)` It stores a single value `x` of generic type A.

```
val x: Option[Int] = Some(10)
val y: Option[Int] = None
val z = None
val empty: Boolean = x.isEmpty
val value: Int = x.get
val valueOrAlternative = y.getOrElse(0)
def quotient(x: Int, y: Int): Option[Int] =
  if y == 0 then None
  else Some(x / y)
```

`x` stores the integer 10.

`y` stores no integer.

The inferred type for `z` is `Option[Nothing]`.

`true` if `x` is `None`, or `false` if a value is stored.

`value` has a 10. An exception `NoSuchElementException` would arise if `value` would be `None`.

Returns the value of `y` if `y` is `Some`, or `0` (valor por defecto) if `y` is `None`.

Returns `None` if `y` is `0` or `Some` with the integer quotient in any other case.

Data deconstruction

- Patterns can be used to manage separately the different cases of a value and, at the same time, to break down a structure into its parts.
- Patterns are the opposite of the constructors (construction vs. deconstruction).
- For example, the patterns for the `Option` type are:
  - `None`, that fits with an empty `Option` value.
  - `Some(x)`, that fits with a non-empty `Option` value, which it is decomposed, hence storing its value into the variable `x`.
- To analyze a value by means of patterns (pattern matching) the **`match case`** instruction is used:

```
def add(x: Int, opt: Option[Int]): Option[Int] = opt match
  case None    => None
  case Some(y) => Some(x + y)
```

- If `opt` matches the pattern `None`, function `add` returns `None`.
- If `opt` matches `Some(y)`, `y` is assigned the value contained by `opt`, and a `Some` is returned with the sum of such a value plus the argument `x`.

Data deconstruction: Fibonacci

- The n-th number in the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, …

```
def fibonacci(n: Int): Long =
 def aux(i: Int): Long =
   i match
     case 0 => 0
     case 1 => 1
     case _ => aux(i - 1) + aux(i - 2)

 require(n >= 0)
 aux(n)
```

Immutable lists

- A value of type `List[A]` is a sequence of values of the generic type A (homogeneous structure) that can be:
  - Empty. It is written as `List[A]()` or `Nil`
  - Start with a value `x` whose type is A (list's **head**) and be followed by another list `xs` whose type is A (list's **tail**). It can be written as `x :: xs`
- Operator :: has a type `(A, List[A]) => List[A]`

```
val list1 = List[Int]()
val list2 = 30 :: List()
val list3 = 20 :: (30 :: List())
val list4 = 10 :: 20 :: 30 :: List()
val list5 = List()
```

An empty list of integers

List of integers (inferred type) whose head is `30` and an empty tail of integers

Head is `20` and `tail` is `30 :: List()`

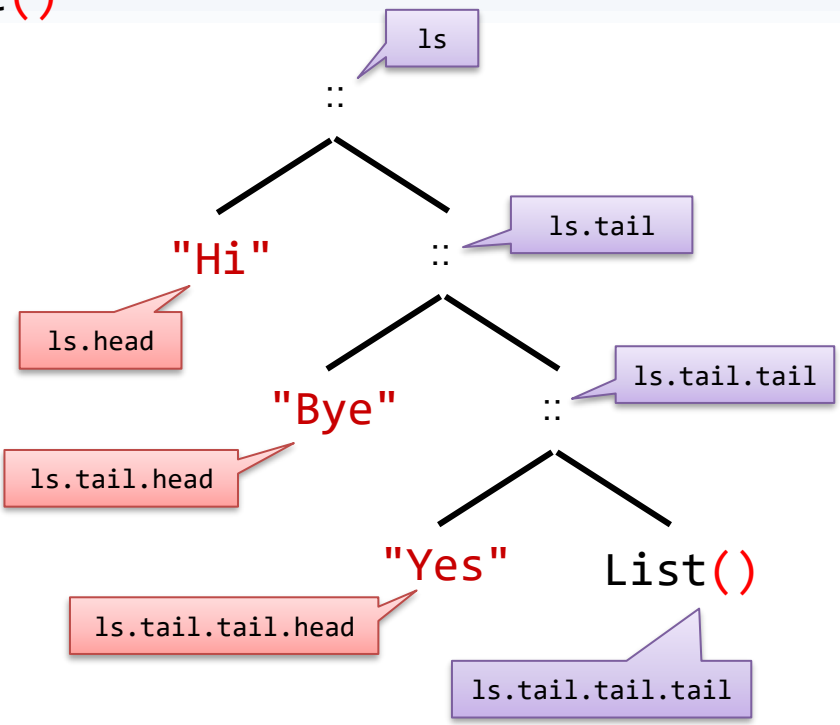
Head is `10` and tail is `20 :: 30 :: List()` (Operator `::` is right associative, so no parenthesis are needed)

An empty list whose inferred type is `List[Nothing]`

Immutable Lists (II)

```
val ls = "Hi" :: "Bye" :: "Yes" :: List()
```

El tipo inferido para `ls` es `List[String]`



- The main methods for lists are:
  - The `isEmpty` method that checks if a list is empty

```
val empty: Boolean = ls.isEmpty
```

  - The `head` and `tail` methods that return the head and tail of a non-empty list

```
val head: String = ls.head
val tail: List[String] = ls.tail
```

A functional list is a recursive structure :
- The base case is the empty list.
- A non-empty list is composed of a first item (the head) and a tail that is, in turn, another list with the remaining items.

Immutable Lists (III)

- Factory methods allow you to build lists following other approaches:

```
val list6 = List(10,20,30)
val list7 = List.range(1,10)
val list8 = List.range(1,10,2)
val list9 = List.fill(3)(true)
val list10 = List.tabulate(5)(i => i * i)
val list11 = List.iterate(1,5)(n => 2 * n)
```

List is `10 :: 20 :: 30 :: List()`

List is `List(1,2,3,4,5,6,7,8,9)`

List is `List(1,3,5,7,9)`

List is `List(true,true,true)`

List is `List(0,1,4,9,16)`

List is `List(1,2,4,8,16)`

- The method `length` returns the number of items in a list

```
val l = list6.length
```

`l` is `4`

- Operator ++ whose type is `(List[A], List[A]) => List[A]` is used to concatenate two lists (it is O(n), where n is the firs list's length)

```
val list12 = list6 ++ list8
```

`list12` is `List(10,20,30,1,3,5,7,9)`

List Patterns

- The constructors for the `List` type are:
  - `List()` or `Nil` to build an empty list.
  - `x :: xs`, where x is an item and xs is another list, to build a list whose head is x and its tail is xs.
- Patterns can be used to identify different cases of lists and, at the same time, decompose them into their components.
- The basic patterns for a List are:
  - Pattern `List()` or `Nil`, that fits an empty list.
  - Pattern `x :: xs`, that fits with a non-empty list. x receives the value of the list´s head, and xs receives the value of the tail.

Next function returns the sum of the squares of the items in the `xs` list. The base case is when the list is empty. The recursive case is when the `xs` list has a head `y` and a tail `ys`.

```
def sunSquares(xs: List[Int]): Int = xs match
    case List()  => 0
    case y :: ys => y * y + sumaCuadrados(ys)
```

- If `xs` matches the pattern `List()`, `0` is returned
- If `xs` matches the pattern `y :: ys`, `y` is assigned the head and `ys` is assigned the tail.

List Patterns (II)

- Other examples of patterns for Lists are:
  - Lists with an exact length.
    - Pattern `List(x)`, that fits with a list with a single item. x will be assigned its value.
    - Pattern `List(x,y)`, that fits a list with exactly two items. x will take the value of the first element and y that of the second.
    - Pattern `List(0,x,y)`, that fits a list with exactly three elements whose first one should be `0`. x will take the value of the second element and y that of the third.
  - Lists with at least a certain length.
    - Pattern `0 :: xs`, that fits with a non-empty list that begins with `0`. `xs` receives the list's tail.
    - Pattern `x :: y :: zs`, that fits a list with at least two items. x will take the value of the first element and y that of the second and `zs` receives the tail of the tail of the list. (List without the first two items).
  - Lists of other structures.
    - Pattern `List((x,y), (z,v))` that fits with a list of tuples that contains exactly two tuples.
    - Pattern `(x,y) :: (z,v) :: us` that fits with a list of tuples that contains at least two tuples.

The map method

- Higher-Order Methods: map

```
val squares = List.range(1,5).map(x => x * x)
```

List is List(1,4,9,16)

```
val evens = List.range(1,5).map(x => x % 2 == 0)
```

List is List(false,true,false,true)

By using the underscore notation, the λ function can be written as:

```
val evens = List.range(1,5).map(_ % 2 == 0)
```

- The map method:
  - Works over a list whose items are of type A.
  - Takes as an argument a function whose type is A => B.
  - Applies the function to all every item in the list and returns a list with the resulting values of type B.

The `filter` method

- Higher-Order Methods : `filter`

```
val greaters = List.range(1,10).filter(_ > 5)
```

List is `List(6,7,8,9)`

```
val evens = List.range(1,10).filter(_ % 2 == 0)
```

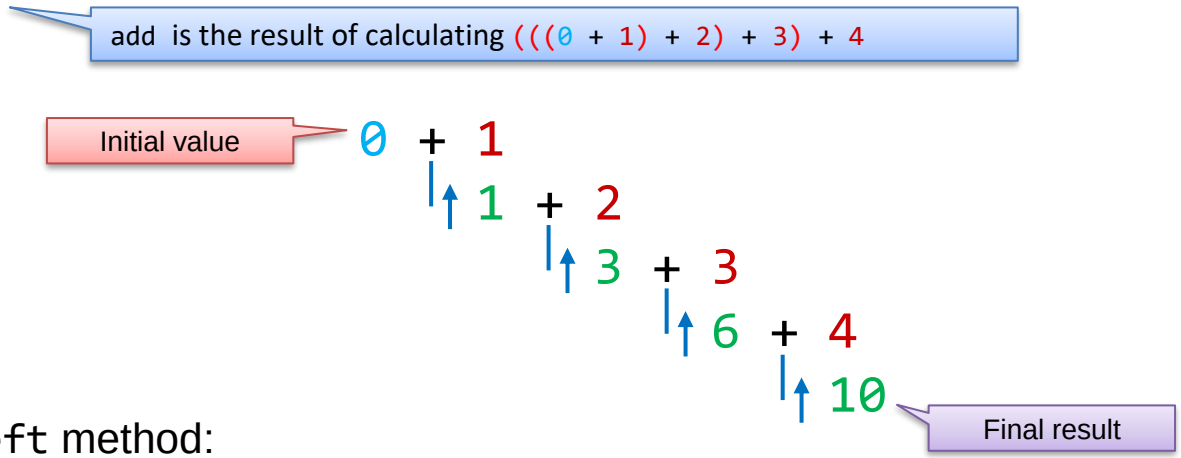
List is `List(2,4,6,8)`

- The `filter` method :
  - Works over a list whose items are of type A.
  - Takes as an argument a predicate whose type A => `Boolean`.
  - Returns a list of type A with the items satisfying the predicate.

The `foldLeft` method

- Higher-Order Methods : `foldLeft`

```
val add = List(1,2,3,4).foldLeft(0)((x, y) => x + y)
```

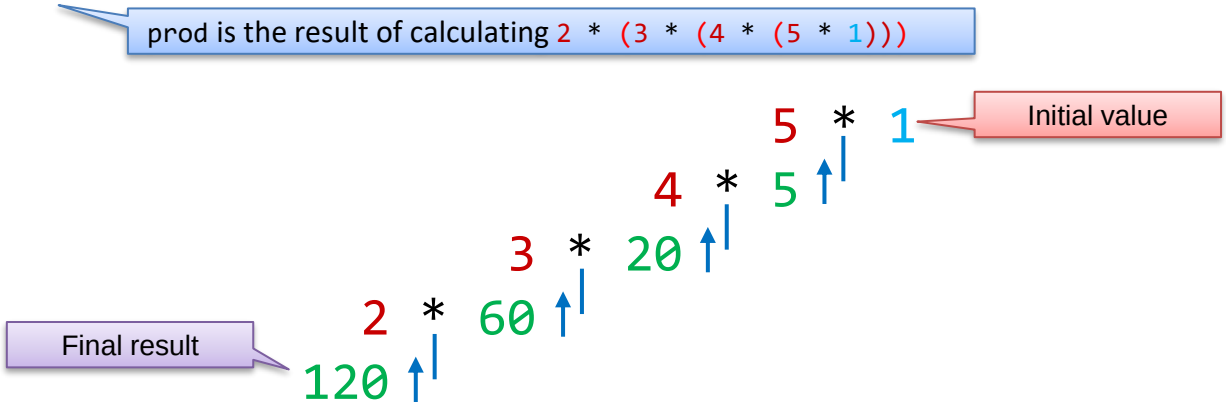


- The `foldLeft` method:
  - Works over a list whose items are of type A.
  - Takes an argument of type `B` (initial value).
  - Takes a function whose type is `(B, A) => B`.
  - Begins with the initial value on the left and apply the function to the result of the previous step. It does the same with each item in the list, from left to right.
  - Returns as a result the last value calculated, of type `B`.

The foldRight method

- Higher-Order Methods : foldRight

```
val prod = List(2,3,4,5).foldRight(1)((x, y) => x * y)
```



- The foldRight method:
  - Works over a list whose items are of type A.
  - Takes an argument of type B (initial value).
  - Takes a function whose type is (A, B) => B.
  - Begins with the initial value on the right and apply the function to the result of the previous step. It does the same with each item in the list, from right to left.
  - Returns as a result the last value calculated, of type B.

for expressions

- For expressions or comprehension lists

```
val evenSquares = for x <- List.range(1,10) if x % 2 == 0 yield x * x
```

- For expressions allow (optionally) filtering the items in a list and then transform them using an expression, returning a list with the results.

- The above example is equivalent to:

```
val evenSquares =
        List.range(1,10).filter(_ % 2 == 0).map(x => x * x)
```

for expressions (II)

- With several generators the Cartesian product may be generated.

```
val cartesian =
      for
        x <- List(1,2,3)
        y <- List('a','b')
      yield (x, y)
```

List is `List((1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b'))`

- Exercise: defines a recursive function that returns all binary strings of size n. For example, those of size 3 are: `"000"`, `"001"`, `"010"`, `"011"`, `"100"`, `"101"`, `"110"` y `"111"`

Bibliography

- [Pilquist et al. 2023] Functional programming in Scala. M. Pilquist, R. Bjarnason, P. Chiusano, M. Odersky. O'Reilly Safari Books, 2023.
- Materiales de la asignatura Informática II del Grado en Matemáticas de la UMA, 2023/24, por J.E. Gallardo y F. Gutiérrez.