L3: Records and Collections

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

September 25th, 2020

Architecture Research Group

SSE, TGGS

Recap

Functional Language

- Now that you installed Scala
  - Let’s formally define what is a functional language

- Program is created by applying functions

- Function definitions are tree of expressions
  - Eventually reduce to a value in most cases

- Function is a first-class citizen
  - We will discuss this in a few weeks

Repetion (Instead of a Loop)

- Recursion is the answer to looping
  - Calling the function itself again → next iteration
- You can write

```
def func_name(v1: Type, …, vN: Type): RetType = {
…
}
```

  - The return type is optional unless your function is recursive

Compound Data

- So far, we have talked about a single data item
  - Number
  - Boolean
  - Conditionals
  - Variables
  - Functions
- Let's look at a way to build up data with multiple parts

Tuple

- A fixed number of items, each can have different type
- Example:
  - (“hello”, 1) will results in a value of type (String, Int)

List

- Lists in Scala and most functional language are front-access lists
- List() makes an empty list
  - Type List[Nothing]
  - We can force a type by saying List[Type]
    - Example: List(): List[Int]
- You can also make a list with elements in it
  - List(1,2,3,1,2)
- You can stick element to the front of the list
  - You will get a brand new list
- Specifically
  - If e1→v, e2→l = [v1, v2, … vn], where e1: T and e2: List[T] then e1::e2 has the type List[T] and represent [v, v1, v2, … vn]

Pattern Matching

- Benefits:
  - Gets a warning if you are missing any cases
  - Gets a warning if you have duplicate cases
  - Most concise, and hopefully more readable
    - Compared to tons of functions …
- Example: You can also use pattern matching to break down tuples in a list
  - Let’s say you have a list of (Int, String)

    ```
    xs match {
    case (number, name)::t => …
    …. // Other cases here
    }
    ```
  - This will break down to the numbers and names for you

Options

- Option is a type
  - Option[T]
- Think of it as Option[T] is either
  - ***None*** that expresses emptiness
  - ***Some(v: T)*** that keeps a value v of type T

Tail Recursion

- The difference here is that the last line of the function is a call to a function
  - Not to itself but to a tail call
  - This is call *tail recursive*
- Benefits:
  - Stack frames can be recycled
  - Compile to a very nice iterative program with no additional state on each stack call
    - Reduce burden on the compiler
- In the previous example, *prod* is the accumulator
  - Accumulate the answer we have so far instead of waiting for the call to return

Mutability

Components of PL

- Syntax: How do you write the language?
- Semantics: What do program mean?
  - I.e., what are the evaluation rules?
- Idioms: What are the typical patterns for using language features to express computation?
- Libraries: What facilities does the language provides?
  - I/O, Data structures, etc.
- Tools: What is provided to make your job easier?
  - A debugger
  - REPL interface

Mutable vs. Immutable

- At this point you probably realize you can modify a list
  - You can append to existing list to create a new list
- What does this mean?
  - Let’s say x is mapped to a value (which can be a List(1,2,3))
  - This x will be forever mapped to this list, and nothing will change x to map to a different list
- Generally, we have a construct to build compound data and accessing pieces of compound data
  - But no construct to mutate the data we built

Mutable vs. Immutable

- Immutable benefits
  - You can guarantee no other code is doing something that make your code wrong (example: no one can modify existing lists)

- Let’s go through a few examples

Example

- def sortPair(p: (Int, Int)): (Int, Int) =
  if(p._1 < p._2) p else (p._2, p._1)
- def sortPair2 (p: (Int, Int)): (Int, Int) =
  if(p._1 < p._2) (p._1, p._2) else (p._2, p._1)
- What are the differences between the two considering:
  - If a pair is immutable
  - If a pair is mutable

- For a language that allows mutable data, the two functions behave differently

Example #2

- def concat(xs: List[Int], ys: List[Int]): List[Int] =
  if (xs.isEmpty) ys else (xs.head)::concat(xs.tail, ys)

- Let’s assume xs = List(1,2) and ys = List(3,4,5)
- What can be the difference if we assume
  - Mutation is allowed
  - Mutation is not allowed

In-class Exercise 4

- Write a function ***def find(xs: List[(Int, String)], key: Int): Option[String]*** that takes in a list of key-value (Int,String)-pairs and returns the string value matching the given integer key. It should return None if nothing matches it.
- Write a function ***def rev(xs: List[Int]): List[Int]*** that takes a list and produces the reverse of the input list. Can you write it as a tail-recursive function?
- Write a function ***def fib(n: Int): Long*** that computes the n-th Fibonacci number in a tail-recursive manner.

Records and More Pattern Matching

Generalizing Compound Types

- Product type: “each of”
  - A value contains values of predefined types
  - Example: Tuple
- Sum type: “One of”
  - A value is one of many types
  - Example: Option
- Recursive: Making self reference
  - A value of type T can refer to a value of type T
  - Example: List

Type Alias

- You can define an alias of a type

- Example:
  - type Person = (String, Double, Int, String)

- This might still be annoying because you need to remember what values should go in which order
  - First entry in the tuple is the name
  - Second entry is the height
  - Third entry is the age
  - I cannot even come up with what should go into the fourth …

Record

- Record addresses the problem we just discussed
- case class Person(name: String, height: Double, age: Int, address: String)
- This make a named record for Person
- To use the record you make, you can:
  - Person("John", 1.80, 30, "Thailand")
    - Notice you need to have the correct order
  - Person(height=1.80, address="Thailand", age=30, name="John")
- You can also bind a named record to a name using val
  - val p1 = Person("John", 1.80, 30, "Thailand")
- You can use the fieldname to access individual field
  - p1.name
  - p1.address

Reference by Name vs. Position

- Notice how you can refer to items in a record by name
- While you can refer to items in a tuple by position

- Different programming language can use either one, or a hybrid approach
  - Java method arguments
    - Caller uses position, callee uses variables
  - Python
    - By position for required arguments and by name for optional arguments

Syntactic Sugar

- Basic idea: Making semantic easier to use

- Example: you can implement a tuple using records
  - case class MyPair(_1:Int, _2: Double)
- We will call this “tuples are syntactic sugar for records”

- Basically syntactic doesn’t introduce a new semantics
  - No new meaning
  - But repackage it to something that looks nicer

Creating Sum Types

- Let's expand our exposure to sum types beyond options
- What if we want to create all arithmetic expressions that involve addition and multiplication

- trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr

Creating Sum Types

- trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr

- Expr is one of the following:
  - A constant with value n, type double
  - A sum of two expressions
  - A product of two expressions

Example

- What if I want to create a rank of playing cards
  - Jack, Queen, Ace, King and all the numbers

- trait Rank
  case object Jack extends Rank
  case object Queen extends Rank
  case object King extends Rank
  case object Ace extends Rank
  case class Num(num: Int) extends Rank

- Notice we mix up both class and object in this sum type

Pattern Matching with Sum Types

- As discussed previously, you can pattern match sum types
  - Example: pattern matching objects
- Let's assume the following for our example

- trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr

Example 1

- What if we want to evaluate the sum type

- def eval(e: Expr): Double = e match {
  case Constant(n) => n
  case Negate(e) => - eval(e)
  case Sum(e1, e2) => eval(e1) + eval(e2)
  case Prod(e1, e2) => eval(e1) * eval(e2) }

Example 2

- What about just printing the expression

- def stringify(e: Expr): String = e match {
  case Constant(n) => n.toString
  case Negate(e) => “-” + stringify(-e)
  case Sum(e1,e2) => stringify(e1) + " + " + stringify(e2)
  case Prod(e1,e2) => "(" + stringify(e1) +")*(" +
  stringify(e2) + ")" }

Example 1+2

- We can combine the two as one object
- Object ExprEval{
  def eval(e: Expr): Double = e match {
  case Constant(n) => n
  case Negate(e) => - eval(e)
  case Sum(e1, e2) => eval(e1) + eval(e2)
  case Prod(e1, e2) => eval(e1) * eval(e2) }

  def stringify(e: Expr): String = e match {
  case Constant(n) => n.toString
  case Negate(e) => “-” + stringify(-e)
  case Sum(e1,e2) => stringify(e1) + " + " + stringify(e2)
  case Prod(e1,e2) => "(" + stringify(e1) +")*(" +
  stringify(e2) + ")" }

}

Default Case

- Similar to a switch case statement, we can have a default case

- case _ => [code goes here]

Pattern Matching Benefits

- Generally making codes look less ugly
- You get warning if you miss any cases
  - Or if you have duplicated cases
- Works for both options and list

In-class Exercise 5

- Implement the following function
- def zip(x : List[Int], y: List[Int]) : List[(Option[Int], Option[Int])] takes, for example, (List(3,2,5), List(6,1,9)) and returns List((3,6), (2,1), (5,9)).
  Hint: you can pattern match on tuples. case (Nil, Nil) is valid.

- def unzip(zipped : List[(Option[Int], Option[Int]))) : (List[Int], List[Int]) takes, for example, (List((3, 6), (2,1), (5,9)) and returns (List(3, 2, 5), List(6, 1, 9)).

Enumeration

Enum

- In Scala, we can use trait for enum
- Example:
- trait Direction
  case object North extends Direction
  case object South extends Direction
  case object East extends Direction
  case object West extends Direction

- Is the same as
  public enum Direction { NORTH, SOUTH, EAST, WEST }
  in java

Complex Pattern Matching

- Extract the first two element of a list?
- val (fst, snd) = xs match { case a::b::_ => (a, b) }
  - Note, this will give a warning and will fail if the list has <2 items

- Use if inside the case
- def quantify(num: Int) = num match {
  case n **if n > 100** => n.toString + " is huge"
  case n **if n > 10** => n.toString + " is large"
  case _ => "small" }

Let’s Rewrite the min function

- Extract the minimum number in a list
- We can do the following:
- def min(xs: List[Int]): Option[Int] = xs match {
  case Nil => None
  case h::Nil => Some(h)
  case h::t => Some(Math.min(h, min(t).get))
  }

Polymorphic Types

- Given a list of integers and a position k, can you write a function nth that ***returns the k-th element in the lis***t (k starts from 0)?
- def nth(xs: List[Int], k: Int): Int =
  if (k==0) xs.head else nth(xs.tail, k-1)
- What if I keep asking for a list of String, Double, etc.
  - This gets annoying
- You can use a polymorphic type for this

Example

- We first declare
- def nth[A](xs: List[A], k: Int) = ???

- Then write the body of nth
- def nth[T](xs: List[T], k: Int): A =
  if (k==0) xs.head else nth(xs.tail, k-1)

- This code expend a list of element, each of type T, and return the k-th element of this same type T

Example 2: zip

- Remember our last in-class exercise? Let’s use the concept of polymorphism for the zip
- def zip[A, B](xs: List[A], ys: List[B]): List[(A, B)] =
  (xs, ys) match {
  case (Nil, Nil) => Nil
  case (x::xs, y::ys) => (x, y)::zip(xs, ys)
  case _ => ??? // should not happen }

- This work for case class, case object too
  - We will get into this later

Fold Operation

Folding

- What if you want to iteratively apply an operation on all elements and accumulate results?

- This operation is called folding

Folding

- Assume: def foo(lst): accum_state = (...initial state...) for elt in lst:
  accum_state = do_magic(accum_state, elt) return accum_state
- You can use fold for this by
- xs.foldLeft initialState doMagic

- This start from the list xs, and then accumulately perform the doMagic function to each element of xs from left to right

Collection

Exceptions

- What if your program run into a rare state such as
  - Accessing an empty list
  - Evaluate 2/0
- Basically the program wants to convey something is wrong
- Exception is a built-in feature of a language to handle these
- In Scala, we can use the throw keyword to raise an exception
- throw new IllegalArgumentException

Result Type of an Exception

- Remember Scala is a strongly-typed functional language
- Exception has a result type
- Example:
- val hdOfList = xs match {
  case h::_ => h
  case Nil => throw new RuntimeException("xs can't be empty") }
- In this case, the exception will by correctly type Int if xs is a list of Int

What to Do With an Exception?

- Ignore → Your program terminate
- Catch and handle it
- def divMod(x: Int, y: Int) =
  try {
  (x/y, x%y)
  } catch {
  case e: ArithmeticException => (0, 0)
  }

Other Uses

- Let's consider this function
- def findLast(xs: List[Int], key: Int): Option[Int] = {

```
def iterFind(xs: List[Int], location: Int): Option[Int] =
xs match {
  case Nil => None
  case h::t => {
    val tailFound = iterFind(t, location+1)
    if (h==key && tailFound.isEmpty) Some(location)
    else tailFound
  }
  }
iterFind(xs, 0) }
```

Other Uses

- What if I call findList(List(1,2,3,2,4,2,5), 2)
- This is going to be a long chain calls

- Solution: We can use exception to jump right out!

Other Uses

- def findLast(xs: List[Int], key: Int): Option[Int] = {

```
  case class FoundIndex(loc: Int) extends Exception
  def iterFind(xs: List[Int], location: Int): Option[Int] =
  xs match {
    case Nil => None
    case h::t => { val tailFound = iterFind(t, location+1)
      if (h==key) { throw FoundIndex(location) }
      else tailFound } }
  try { iterFind(xs, 0) } catch {
  case FoundIndex(loc) => Some(loc)
  }
}
```

Before We Leave Today

Collections Can Be Versatile

- Scala’s collections come with library method
- Example: let’s assume val L = List(1,2,3)
- You can do L.length to get the length of this list
- You can do L.exist(predicate) to check if the matching predicate exist
- You can map a function to all elements using map
  - For example, you can multiply all elements by 2 using L.map(x=>2x)
- You can filter out elements
  - L.filter(x=>x<2)
- Add all of them using L.sum
- Drop elements
- More info on scaladoc

Assignment 1 Is Out

- Due in 2 weeks
- If you are done with in-class exercise
  - Feels free to work on this during the rest of this class slot