L4: OOP, First Class Functions, Continuation and Closure

Rachata Ausavarungnirun

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

October 2nd, 2020

Architecture Research Group

SSE, TGGS

In-class Exercise 6

- def unzip(xs: List[(Int,Int)]): (List[Int], List[Int]) reverses what zip does. Make it so that it's polymorphic. The input can be any List[(A, B)].
- def countWhile[T](xs: List[T], key: T): Int that counts the number of times key repeats itself in the prefix of xs.
- def topK(xs: List[Int], k: Int): List[Int] that tallies the elements of xs and return elements with the top k frequencies (if there are ties, break ties in any way you like). Look at Scaladoc for inspiration.
- Make a sum type called Dessert, which can be one of the following: – Pie(kind: String), – Smoothie(fruits: List[String]), – Cake(toppings: String)
- Then, write a function def isLiquid(what: Dessert): Boolean

Let’s Create a Rational Number

- Idea 1: Use a pair
  - type Rational = (Int, Int)
  - def add(p: Rational, q: Rational) = (p, q) match {
    case ((np, dp), (nq, dq)) => (np*dq + nq*dp, dp*dq)
    }
  - def toString(p: Rational) = (p, q) match {
    p.toString + "/" + q.toString
    }
- Idea 2: Use a record
  - case class Rational(n: Int, d: Int)
  - def add(p: Rational, q: Rational) = Rational(p.n*q.d + q.n*p.d, p.d*q.d)
  - def toString(p: Rational) = p.n.toString + "/" + p.d.toString

Using the Class

- Let’s use a class to define a rational number
- class Rational(n: Int, d: Int) {

```
class Rational(n: Int, d: Int) {
        def numer = n
        def denom = d
}
```

- Instantiation in Scala
  - new
- val r = new Rational(3, 4)
- Accessing r can be done by using *r.numer* and *r.denom*

Implementing an Add

- This can be done so it becomes a class method
  - def add(that: Rational) = new Rational(this.numer*that.denom + that.numer*this.denom, this.denom*that.denom)

- def mult(that: Rational) = new Rational(numer*that.numer, denom*that.denom)
- override def toString = numer + "/" + denom

Public and Private

- private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  private val g = gcd(n, d)
  def numer = n/g
  def denom = d/g
  require(d>0, “denominator must be positive”)

- By default, def is public

Constructor

- We can define a constructor by adding aux. constructors
- def this(n:Int) = this(n, 1)

- Notice how we use “this” to self-reference
- Example:
- def less(that: Rational) =
  numer * that.denom < that.numer * denom
- // This could have been: this.numer * that.denom < that.numer * this.denom

Overloading Operators

- Unlike an Integer, adding a rational class is different
  - You cannot just call x+y
  - You ended up having to define r.add
- Alternative: overloading the “+” operator
  - Operators are treated like a function, you can define it

- def + (r: Rational) = …
- def – (r: Rational) = …
- def unary_- = …
- Keep in mind that operators have precedence rule
  - Overloaded operators keep the same rule

Abstract Class

- What if we want to make an abstract class?
  - Q: What is an abstract class?
- Let’s say we want to create the following things:
- An IntSet, where it collects a set of Integers
  - add(x: Int): IntSet – produce a new set taking the union of this set and {x}.
  - has(x: Int): Boolean — ask if x is a member of this set
- How can we specify the interface?

Abstract Class: Interface

- Create an abstract class
- abstract class IntSet {

```
        def add(x: Int): IntSet
        def has(x: Int): Boolean
}
```

- Then, we can implement this class later

Abstract Class: Implementation

- Example: Implement this IntSet using a linked list
- class Empty extends IntSet {
  ```
  class Empty extends IntSet {
          def has(x: Int) = false
          def add(x: Int) = ... // new NonEmpty(x, new Empty)
  }
  ```
- class NonEmpty(elt: Int, other: IntSet) extends IntSet {
  ```
  class NonEmpty(elt: Int, other: IntSet) extends IntSet {
          def has(x: Int) = (x==elt) || (other has x)
          def add(x: Int) = new NonEmpty(x, this)
  }
  ```

Abstract Class: Implementation

- Empty and NonEmpty extend the class IntSet
- Both conforms to IntSet
- IntSet is the superclass to Empty and NonEmpty
  - Vice versa, Empty and NonEmpty are the subclasses
- Everything has an Object as a superclass
  - This includes your REPL statements
  - This means you can override the implementation
    - val on top on existing variables

Limit Copies to One

- From our example, the Empty set should really have one copy, right?
  - This is pretty easy to fix using static class in other languages
- In Scala, this is also an easy fix using a singleton ***object***
- object Empty extends IntSet {

```
object Empty extends IntSet {
        def has(x: Int) = false
        def add(x: Int) = new NonEmpty(x, Empty)
}
```

- This define an object called Empty, no other instances of this object can be created
- Empty evaluate to itself (it is a value)

In-Class Exercise 7

- Recreate an object for the Expression type with we been using, with the following traits
  - trait Expr {
    def +(that: Expr) = [Fill in this blank]
    def *(that: Expr) = [Fill in this blank]
    def unary_- = [Fill in this blank]
    def toVal(implicit ctx: Map[String, Double]): Double
    }
- It should have the following methods
  - case class Var(name: String) extends 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
  - Each of these should implement its version of ***toVal***

First-class Functions

Recap: First Class Function

- Functions become values
- Conceptually, this allows you to pass functions in, and return a function

- Example: Repeat a function n times
  - def nTimes[A](f: A => A, n: Int, x: A): A =
    if (n==0) x else f(nTimes(f, n-1, x))

Examples: Functions as Inputs

- Let's define:
  def triple(x: Int) = 3*x
  def addTwo(x: Int) = x+2
  def doTail[T](xs: List[T]) = xs.tail

- What does these do?
  nTimes(triple, 7, 11)
  nTimes(addTwo, 4, 9)
  nTimes(doTail, 2, List(3,5,2,4,9,7))
  nTimes(doTail[Int], 2, List(3,5,2,4,9,7))

Examples: Functions as Outputs

- ```
  def tripleNTimes(n: Int, x: Int) = {
    def triple(x: Int) = 3*x
    nTimes(triple, n, x)
  }
  ```

- ```
  // use the shorthand form for defining a function
  def tripleNTimes(n: Int, x: Int) =
    nTimes((x:Int) => 3*x, n, x)
  ```

Scala: Methods vs. Functions

- When we write def inc(x: Int) = x+1
  - This is not really a function
  - def with parameters is a method
- In Scala, method can be polymorphic
- Also in Scala, functions are never polymorphic
  - They will have a type
- inc _ gives a functional form, it takes an Int, and will return an Int

Types

- For now, let's assume functions are polymorphic
- def nTimes[A](f: A => A, n: Int, x: A): A =
  if (n==0) x else f(nTimes(f, n-1, x))
  - This has the type ((A => A), Int, A) => A
    - What does this mean?

- In this same example, A is a placeholder for a type
- But, these functions does not have to be polymorphic
  - def timesUntilZero(f: Int => Int, x: Int): Int =
    if (x==0) 0 else 1 + timesUntilZero(f, f(x))

Reducing the Function

- Consider this example
  - if ((x*y+2 < 10) == true) true else false
- Rewrite once
  - if (x*y+2 < 10) true else false
- Rewrite again to
  - (x*y+2 < 10)

Reducing the Function: Example 2

- Can I rewrite the following?
  - nTimes(doTail[Int], 2, List(3,2,1))

- nTimes((xs: List[Int]) => xs.tail, 2, List(3,2,1))

- nTimes[List[Int]](_.tail, 2, List(3,2,1))

More Abstraction

- Consider this example
  - def sillyLottery(f: Int => Int, n: Int) =
    if (f(n)%2 == 0) {
      (x: Int) => x/2
    } else {
      (x: Int) => 2*x+1 }
- What is the type?
  - **((Int => Int), Int)** => **(Int => Int)**
  - If we give Int => Int and one Int, we will get Int => Int
  - Which we can bind to a variable
- Let's consider val magic = sillyLottery(x=>3*x-9, 25)
- What is magic(21)?

Scala with I/O

- You can import scala.io.Source to deal with I/O

Example

- What if I want to count the number of word in a file?

```
import scala.io.Source
object SimpleWordCount extends App {
  def countPerLine(line: String): Int =
    line.split("\\W+") .length
  val wordsPerLine =
    Source.stdin .getLines.map(countPerLine).toSeq
  val lineCount = wordsPerLine.length
  val wordCount = wordsPerLine.sum
  println(s"lineCount: $lineCount")
  println(s"wordCount: $wordCount")
}
```

Continuation

Continuations

- So far, all our functions return something
- You can also call a new function at the end
  - This is called the continuation passing style (CPS)

- This allows you to make every function a tail call
- Let’s use a sum of all integer as an example
  def sum(L: List[Int]): Int = L match {
    case Nil => 0 case x::xs => sum(xs) + x

Continuations

- Tail call version:
- sum_tc(L: List[Int]): Int = {

```
  def sumHelper(L: List[Int], a: Int): Int = L match {
    case Nil => a
    case x::xs => sumHelper(xs, a + x) }
  sumHelper(L, 0) }
```

Continuations

- Continuation version
- def sum_cont(L: List[Int]): Int = {
  def sumHelper(L: List[Int], **K: Int => Int**): Int = L match {
  case Nil => **K(0)**
  case x::xs => sumHelper(xs, (r: Int) => **K(r + x)**) }
  sumHelper(L, **(x: Int) => x)** }

Example 2: Binary Tree

- Fundamentally, a binary tree can be defined recursively
  - A node can be
    - Empty
    - A node with two sub-tree
- Let’s make the trait Tree
- sealed trait Tree
  case object Empty extends Tree
  case class Node(**left: Tree, key: Int, right: Tree**) extends Tree

Example 2: Binary Tree

- Making a tree can be done by
- val t = Node(Node(Empty, 2, Empty), 5, Node(Node(Empty, 6, Empty), 7, Node(Empty, 9, Empty)))

- What does this tree looks like?

Example 2: Binary Tree Traversal

- ```
  def walkInorder(t: Tree): List[Int] = t match {
    case Empty => Nil
    case Node(l, k, r) => walkInorder(l):::(k::walkInorder(r))
  }
  ```

- What if we want to use continuation?
  - We need to pass done the functions to call at the end
  - What should that function do?
  ```

Example 2: Binary Tree Traversal

- def walkInorder(t: Tree): List[Int] = {
  def contWalk(t: Tree, K: List[Int] => List[Int]): List[Int] = t match {
    case Empty => K(Nil)
    case Node(l, k, r) => contWalk(l, leftList => {
      contWalk(r, rightList => K(leftList:::(k::rightList)))
     })
    }
  contWalk(t, (r: List[Int]) => r)
  }

Currying

Currying

- Instead of accepting parameters normally, accept through a sequence of functions
- def sortedUncurry(x: Int, y: Int, z: Int) = x <= y && y <= z
- val sortedTriple = { (x: Int) => (y: Int) => (z: Int) => x <= y && y <= z }

- Currying version:
  - def sortedTriple **(x: Int) (y: Int) (z: Int)** = x <= y && y <= z

Currying

- Benefits:
  - You can stage the function
    - Parts of the execution can run as soon as the values are ready
  - Maps well with dataflow model
  - This can allow the compiler and the hardware to be faster
    - Eliminate data dependency as soon as possible

- Actual efficiency: It depends
  - Compiler is very smart nowadays
  - Run a profiler to test the two formats

States and Mutable Variables

- We assumed variables are immutable
  - This is annoying in some cases
  - What if we need to store a state

- **State:** the intermediate steps that need to be stored
  - Real hardware also needs the concept of state

- So, many functional languages have mutable variables

- Benefit of mutable variables
  - Allow programmer to keep the state

Declaring Mutable Variables

- var x = value
- Example: implementing a while loop
  - Using currying and mutable variables
- def my_while (condition: => Boolean) (block: => Unit): Unit = {

```
  if (condition) {
    block
    my_while (condition) (block)
  } else ()
}
```

Before We Leave Today

In-class Exercise 8

- Implement fibonacci recursively in the continuation-passing style

- Preorder traversal: visit root first, then left, then right. Write preorderWalk in CPS style