Scala: Language Explore

Zoe Nacol Kyle Dymowski Charles Tandy Anthony Nguyen

December 7, 2015

1 Introduction to Scala

Scala is both an object-oriented and functional language [1]. A portmanteau on "scalable" and "language", Scala was created based upon criticism of Java. Running on Java Virtual Machine, language interoperability can be seen between Java and Scala. Both Java and Scala classes and libraries can be mixed. Despite the conventional syntax and it being based off of Java, Scala supports first class functions, immutable data structures, and a general preference of immutability over mutation.

2 Scala Features

2.1 Concepts Learned in Class

"At the root, the language's scalability is the result of a careful integration of object-oriented and functional language concepts" [1]. Scala can be treated as both a scripting language and "workhorse" language. Because of this, Scala supports many useful features, some of which were covered in CSCI 400.

Pattern Matching Scala supports pattern matching, which we saw when learning Haskell. Pattern matching offers an alternative approach to designing functions. Instead of using a series of if-else and instance-of statements in Java, pattern matching proves a better solution because it can match any data type. In Scala specifically, aliases can be created like the below example:

```
1          address @ Address(\_, \_, "Paris", "France")
```

Type Inference Scala also supports type inference, as seen in Haskell. This allows the programmer to omit certain type annotations. "It is...often not necessary...to specify the type of a variable, since the compiler can deduce the type from the initialization expression of the variable." This can also be applied to return types of methods [6].

String Interpolation Scala makes use of string interpolation to create strings of data. This process is used to directly embed variables into string literals. It makes use of three methods of string interpolation. This first is s where string literal allows the usage of variables directly in the string. Second, is f which allows the creation of simple formatted strings similar

to printf in PHP. Third, is raw which is essentially the same as s, but with no escaping of literals in the string [5].

Other Other notable features of Scala seen in class include lazy evaluation, currying, higher order functions, and the concept that functions are objects. There are also several features present in Scala but not Java. Some of these include operator overloading, optional parameters (as seen in Haskell), and no checked exceptions [7].

2.2 Concepts Not Learned in Class

Value Classess In Java there is no difference between the integer type, it can be used through the program for counters, an entity identifier, or a number in an arithmetic expression. In most cases these Integers have nothing to do with each other. It is a common mistake to compare them, do arithmetic operations on them, or pass the wrong one into a function as a parameter. A value class allows a new level of type checking that looks at the code rather than relying on the person doing the coding. A value class works like a wrapper class that gives the ability to enrich objects with a precise type without allocation overhead.

However, limitations stop values classes from being universally used:

- primary constructor for the value class must have exactly one val parameter whose type is not a class [3]
- class may not have specialized type parameters.
- class may not have nested or local classes, traits, or objects
- class may not define a equals or hashCode method.
- class must be a top-level class or a member of a statically accessible object
- class can only have defs as members. In particular, it cannot have lazy vError! Hyperlink reference not valid.als, vars, or vals as members.
- class cannot be extended by another class.

Implicit Classes Scala allows for extension methods to be added onto another type by introducing implicit classes. The main purpose of an implicit class is to make the process more concise. The method definitions must be declared where the scope allows for method definitions. This works by the implicit method mimicking the constructor of the class [4].

Parallel and Concurrent Programming Parallel and concurrent programming are performed in scala through the use of *Futures* and *Promises*. *Future* is an object for a value not made yet, a placeholder. When the value is created, then it is given to *Future* to be completed. Note: *Future*[T] is the upcoming object; *future* is the method that handles and creates *Future* objects. *Promises* can make *Future* objects.

Execution Context An *ExecutionContext* performs some computation based on the results obtained by *future* and *promise* and properly performing computations at the right time under the desired result. ForkJoinPool manages threads for the ExecutionContext, helping in asynchronous computation. (EC performs some computation in a thread, like an executor, based on the results of *future* and *promise*.)

Futures Futures are, as stated, objects for values that have yet to be created but will be made eventually. They have an initial state of being not completed, which will not change up until the *Future* obtains the result of some computation. Once a *Future* has been completed, it cant change back. (Cant change a *Future* when it is already been made).

When a *Future* has been completed, it will either be a success with the expected value or a failure because of some exception due to some error. However, the *Future* will still have that exception, letting it be used however it needs to be.

The *onComplete, onSuccess, and onFailure* methods uses a given callback and executes it on the resulting *future*, preventing the use of blocks which is the goal. onComplete takes all results (as long as the *Future* has been completed) and executes the proper action based on the *Future* being a Success of Failure. onSuccess only deals with success cases; onFailure deals with a specific error given due to failure.

Multiple callbacks are ordered by an appropriate *ExecutionContext*. Using functional composition can be used as an alternate to using callbacks.

Promises *Promises* are another way to make *Futures*. This is done by first making a *future* by a *future* function on the *promise*, and then taking some computed value and using it to fulfill a *promise* and complete the *future*. *Promises* can only be completed once, if it succeeded or failed, then it stays that way and cannot be changed.

3 Demo Program

For the program, we chose to create n-queens in Scala. We chose n-queens because it allowed us to showcase some of the features of Scala in a non trivial way without being overly complicated. Scala by example was used as a reference for the program.

The function placeQueens was used to recursively place all of the queens. We used lists of tuples to store the queens. We made use of pattern matching using match to check the current column n. This was used to establish a base case for the recursive call. We then used yield to prepend a queen to a list of queens after checking if a queen placement was valid in relation to the other queens.

```
1       def placeQueens(n: Int, size: Int): List[List[(Int, Int)]] = n match {
2         case 0 => List(Nil)
3         case \_ => for {
```

```
            queens <- placeQueens(n -1, size)
            y <- 1 to size
            queen = (n, y)
            if (isSafe(queen, queens))
         } yield queen :: queens
      }
```

To check for valid moves we created a method isSafe which takes a tuple to place and a list of tuples representing the placement of the existing queens. In this method we used a conditional for loop that would execute the code returning false if any of the conditions fail for the comparisons with the existing queens.

```
      def isSafe(queen: (Int, Int), others: List[(Int, Int)]): Boolean = {
         for ( q <- others
            if q.\_1 == queen.\_1 ||
            q.\_2 == queen.\_2 ||
            (q.\_1-queen.\_1).abs == (q.\_2-queen.\_2).abs) {
            return false
         }
         return true
      }
```

After running the recursive function and returning the lists of tuples we were able to print out a graphical display of all of the valid placements. This portion was relatively trivial and therefore needs no explanation.

4 References

1. http://www.scala-lang.org/what-is-scala.html
2. https://en.wikipedia.org/wiki/Scala_%28programming_language%29
3. https://ivanyu.me/blog/2014/12/14/value-classes-in-scala/
4. http://docs.scala-lang.org/overviews/core/implicit-classes.html
5. http://docs.scala-lang.org/overviews/core/string-interpolation.html
6. http://www.scala-lang.org/old/node/127
7. http://www.scala-lang.org/docu/files/ScalaByExample.pdf