Jared Girouard – SEC01 (NUID 002981834)

Big Data System Engineering with Scala  
Fall 2022   
Assignment No. 5



**-List of Tasks Implemented**

This assignment had us continue our work on the movie database by implementing the following methods:

* Map2, Map3, Map7
* Lift, Lift2, Lift3, Lift7
* Invert2, Invert3, Invert4
* Uncurried2, Uncurried3, Uncurried7

Each category of method had the same functionality but operated on a different number of parameters. Additionally, a MovieProtocol was created extending the defaultJsonProtocol to format how our Movie class is serialized to Json. Finally, the testSerializationAndDeserialization method was implemented to test that our Movie class can properly be serialized to Json and then deserialized back to match our input.

**-Code**

**Map**

Map is a method that we have seen in class and other assignments, but we have usually been dealing with one parameter in the first parameter set and a function to convert from one type to another. Here, we needed to implement map for 2, 3, and 7 parameters in the first parameter set. Implementing this was fairly straightforward since we were provided with a function that returned an R and we wanted to return a Try[R]. To accomplish this, I just wrapped the call to that function in a Try and passed in the parameters. The process was the same no matter the number of parameters passed in. For map7, I just had to pass 7 parameters into the provided function.

![A screenshot of a computer

Description automatically generated with medium confidence]()

**Lift**

The next group of methods to implement was lift. Lift took a function that transformed one type to another T=>R. Our goal here was to return a function that transformed a Try[T]=>Try[R]. To implement this, I called map using the input to the method as the first parameter set and the function f as second parameter set to be the mapping function. The result was a function that transformed a Try[T] => Try[R]. I followed this same idea for the 2, 3, and 7 argument versions of Lift. I called the map method for the appropriate amount of arguments that I had, so for 3 input arguments, I called map3(\_, \_, \_)(f) passing in the 3 arguments in order.

![Text

Description automatically generated]()

![Text

Description automatically generated]()

**Invert**

The invert method is the method that was probably the most difficult for me to implement. The invert method returns a function that has the same result as the original, but the parameters are all in reverse order. For instance, invert2 takes in a curried function T1=>T2=>R and returns a curried function T2=>T1=>R. To implement this, we could not use the underscore character to represent the input because the compiler infers an order, and we are trying to reverse that order. To get around this, I assigned the parameters to variables (x:T2) and (y:T1). From there, I can refer to them in the order that I want. Then I called the passed in function f to get the R the function is expecting as a result. The complete function looks like (x:T2) => (y:T1) => f(y)(x). The same exact idea was followed for the invert3 and invert4 just with more parameters.

![A screenshot of a computer

Description automatically generated with medium confidence]()

**Uncurried**

The uncurried2 method took in a curried function that had 3 parameters T1, T2, T3 and returned an R, T1=>T2=>T3=>R. The goal of the uncurried2 method was to get the same result as the original function but with 2 of the parameters uncurried, (T1, T2) => T3 => R. To implement this, I assigned the parameters to variables again and just structured the function to look like the expected return function. I called the passed in function f, to get the result “R” and the function looked like this (x:T1, y:T2) => (z:T3) => f(x)(y)(z). If the resulting function were to be used in the future, it would be called f(x,y)(z). The same idea was followed for uncurried3 and uncurried7 except they uncurried 3 and 7 parameters respectively.

![Text

Description automatically generated]()

**Serialization**

The serialization was fairly straightforward but was something we hadn’t talked about in class yet. The goal here was to fill in the MoviesProtocol object which extended the DefaultJsonProtocol to specify how our custom classes should be serialized to json. This was done through the use of implicit vals. JsonFormat was called for each case class in our Movie file. The apply methods for each case class were explicitly called since they all had companion objects within the file. From there, I imported MoviesProtocol.\_ into the IngestibleMovie so it could be used. I then implemented the testSerializationAndDeserialization function which took in a Seq[Movie] and required us to convert the input to json and then back to a Seq[Movie]. We then had to check if the converted Seq[Movie] matched the original input. To do this, I just called toJson on the input and afterwards called convertTo[Seq[Movie]] to convert it back to its original format.

![A screenshot of a computer screen

Description automatically generated with medium confidence]()

**-Unit tests**

**![Text

Description automatically generated]()**

**![Text

Description automatically generated]()**

**- Result**

After implementing all of the function described in this document all of the unit tests passed indicating the implementations were correct. This assignment was great practice manipulating inputs to create new functions based off of existing ones.

Link to repository: <https://github.com/JaredGirouard/CSYE7200>