Jason Vettese

CS 421: Programing Languages and Compilers

Final Project

8 Aug 2020

The Usefulness of Scala: A Multi-Paradigm Programming Language

Scala is a multi-paradigm programming language that shares characteristics of both object-oriented languages and functional languages. Object-oriented (O-O) programming languages revolve around the idea of an "object," and a very common example would be Java. O-O programming many offer built-in benefits such as encapsulation and inheritance. Functional programming languages, on the other hand, relies on applying functions to their arguments. Functional programming is great at lazy evaluation and higher order functions. Scala blends the best parts of both types of languages to create an effective new approach to a multi-paradigm programming language.

Encapsulation is a paradigm that hides data from certain other members in the code. In an object-oriented language, like Java, this can be achieved by making a variable private, and creating mutators and accessors to manipulate the variable. Some functional languages share a similar concept, but it is not as user-friendly. For example, Haskell supports encapsulation by creating different modules and exporting certain functions. However, the functional approach is not equivalent to its O-O counterpart. In Haskell, all data are immutable. This means that when you define `x = 1`, you can't reassign `x` to be anything but `1`. Encapsulation is very useful when trying to manipulate variables to change their values. Scala can support encapsulation through the use of its mutable variables.

Scala supports both mutable and immutable variables. A variable can be declared as immutable with the keyword `val`. A mutable variable is made with the keyword `var`. The ability to have both mutable and immutable variables can be quite useful in practice. Immutable variables guarantee a result to be the same and are naturally thread safe. This can lead to more correct code and less bugs. Java has support for immutability which can be achieved with the `final` tag. By

default, Java classes and variables are mutable. Unlike Java, Scala variables are preferred to be immutable because of the safety associated with it.

Object-oriented languages contain a paradigm called inheritance. This can help maximize code reuse in large software projects. Since Scala has object support, it also has support for inheritance. Haskell, however, does not have native support for inheritance, or at least not in a manner that is as user friendly as an O-O language since it is a functional language. In Java, a class `A` can have a method `print()` that just prints to console. Class `B` can extend class `A` and it will already have the `print()` method defined from its parent class `A`. The syntax in Scala is very similar to Java. On the other hand, Haskell does not allow this behavior. Instead of classes and objects, Haskell supports types and data. There is not a native way to create "sub-types" or whatever the Haskell equivalent to class inheritance would be called. The `deriving` clause for `data` may be the closest thing Haskell has to inheritance. The ability for Scala to support inheritance is advantageous in code reuse and software design.

Unlike most O-O languages, Scala has native support for pattern matching. In this class, CS 421, Haskell's pattern matching was used quite frequently in the machine problems. In fact, the Scala syntax is very similar to Haskell pattern matching. Java does have some sort of pattern matching with the keyword `instanceof`, but it is very inefficient to make that call. Scala is able to efficiently pattern match, similar to functional programming languages, which gives it an advantage against most object-oriented languages.

One advantage of functional programming languages over o-o is that functional languages have great support for lazy evaluation. Lazy evaluation can be very powerful when computing large values or long lists that are not needed entirely or right away. For example, if someone wants to compute the 213th number of the Fibonacci sequence, in an object-oriented language, they would most likely create a method with a parameter that is used as the index of the Fibonacci sequence, then they would iterate 213 times through the sequence. In Haskell, this can be done lazily by creating a function that produces an infinitely long list of the sequence then by calling `take` on the function. Scala has this same support and uses the same keyword; the only catch is that it must be an immutable variable and be declared as `lazy val`. The full implementation of this Fibonacci lazy evaluation can be seen in the code repo. "Why Functional Programming

Matters" calls lazy evaluation "one of the most powerful tool *[sic]* for modularization in the functional programmer's repertoire." Lazy evaluation is great when dealing with non-deterministic programming.

Higher order functions are staple of functional programming languages. They take functions as parameters and return functions. In Haskell this would be the function `map` or `fold`. Object-oriented languages, namely Java, have limited if not any support for higher order functions. This could be because of the lack of support for proper type for function literals, or the issue with primitive values in the type system, that is the case in Java. Scala does offer built-in support for higher order functions. This can be useful in many cases such as the `inclist` problem from the first MP. Basically, the function takes in a list and maps a function `x + 1` to all the elements of the list. This is a very declarative way of programming and can make it easier to read the code. There are, of course, numerous ways to circumvent higher order functions. An obvious approach would be to use recursion and iterate through the entire list. However, when applying partial functions, higher order functions can be a more intuitive way to solve the problem. Newer versions of Java are starting to adopt partial function applications, but it is still more cumbersome than Scala or functional languages. Scala can effectively imitate a functional language by adopting the ability to use higher order functions from within an o-o context.

There are many programming paradigms that make a language useful. Whether it is inheritance in object-oriented languages, or pattern-matching in functional languages. Scala provides a unique blend of functional and o-o programming. It can handle encapsulation and inheritance as easily as Java. Scala also can take full advantage of pattern-matching, lazy evaluation, and higher order functions that make functional programming languages so appealing. Scala is truly versatile and is a great language to learn many of the programming language concepts taught in CS 421. A wiseman once said "Scala would be the result if Haskell and Java had a child together."

Sources:

Why Functional Programming Mattes – John Hughes

https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

Advantages of using the object-oriented paradigm for designing and developing software – E. Post

https://pdfs.semanticscholar.org/0c3b/98172de01f22f1694bd582dec6163f614a95.pdf

Why Haskell matters – HaskellWiki contributors

https://wiki.haskell.org/index.php?title=Why_Haskell_matters&oldid=60185