**Use of Haskell**

There has been an increasing tendency to move from typical object oriented programming, in the software development industry, to a functional programming paradigm. Functional programming has its basis on math as it takes several concepts from Category Theory and typed lambda calculus. There are several advantages in using functional programming. That’s the reason why we choose Haskell over other languages that could fit for a similar purpose and it is going to be further explained in the context of Hadoop and the inherent advantages of Functional Programming.

Hadoop is a tool for processing a large quantity of data in parallel way using a simple programming model with two well defined steps known as the map and reduce steps. Hadoop takes over the responsibility of distributing the data around several nodes to be processed in a parallel way. In order to use Hadoop, we would usually need to implement a program in Java that internally will have a Mapper and a Reducer classes implemented. This forces anyone that would want to use Hadoop to learn Java and follow every restriction or hole that the language could have for certain purposes. Fortunately, Hadoop Streaming is a Hadoop Interface that basically enables the user to upload a two binary files known as mapper or reducer for the map and reduce step in Hadoop. There are some constraints that Hadoop Streaming imposes:

* The mapper and the reduce must be able to read from the standard input of a system as well as to print to the standard output of the system
* Additionally, the mapper’s output must be the reducer’s input.

These feature allows a big scientific community, with no background in software developing, to use Hadoop using the programming language of their choice. This means that quick scripting languages like Ruby or Python can be used to develop the mappers and reducers and use their respective interpreter to execute them, it also means that can look to other alternatives in order to develop the mapper and reducer files for Hadoop. Since we can choose the language that fit the best with our problem.

**Enter Haskell**

Haskell is a functional language that has the majority of the most representative features from functional programming languages as well as typed calculus lambda based languages:

* Declarative: This feature means that the user has to writes the code in a “what” should be done instead of “how”, this abstracts the programmer the unnecessary complex small parts and leaves them to the compiler or to other libraries. The reader can think in a Java program that is composed in several steps where an instruction is executed once the previous was executed, like the described in a Von Neumann machine. In a declarative way the majority of those steps are abstracted for the compiler leaving the programmer only to concern about the actual problem instead of how the machine works. It benefits largely on what a functional paradigm for programming means and as we will see below the compiler plays an essential role in that optimization thanks to other features like pure functions. Therefore, it is easier to extend over the time due to more expressiveness and avoidance of unnecessary complexity in small parts.
* Statically Typed: Every expression in Haskell has a type which is going to be determined at compiling time, if the types do not match then the code will not compile. Here is one of the strongest parts of Haskell, its type system. Haskell’s type system is different from other type systems like Java’s or C++. Unlike Java’s type system, Haskell’s is not constrained by Object-Oriented programming concepts, allowing it to have Algebraic Data Types as well as some of Category Theory concepts like Monads or Functors in a very simple way due to its Typeclasses and pattern matching. This means that the programming paradigm could be different to what the reader could be used to, but it is easily understandable and more expressive due to the declarative syntax and pattern matching. Finally, the type system is not only a guarantee to catch many errors at compiling time but to build a more expressive software making a less error prone code.
* Type Inference: The user does not have to explicitly declare the type of every expression in Haskell. The compiler will determine the type of an expression and the types will be unified, this means that the compiler will infer the most generalized type possible for an expression. The most popular compiler for Haskell, the GHC uses the Hindel-Minler algorithm for the inference, the algorithm supports several features like Polymorphism and Generalized Algebraic Data Type through extensions for the language. However, the programmer is able to explicitly declare the type of a function. For example, “addTwo x y = x + y” will be inferred to be take any two arguments that implement the typeclass Num, that could be an Integer or a Float and even a Date that has that typeclass (The reader can think of a typeclass as an interface in Java).
* Purely functional: Functional languages generally have functions that cannot keep state, ergo variables values could not be mutated over the time. Functions in Haskell can’t change state either and they are pure in a mathematical sense, they cannot change state in the real world. Tasks like writing to the console or reading from it are considered functions that require to conserve and change state. Haskell has its idioms and construct to deal with this like an IO monad to express or describe what it should do by pure code. This enforces the separation from actions that could have side effects that change the world and pure functions. Therefore, usually in functional programming, tasks are decomposed into small pure functions that will evaluate others and tend to be very specific to where a side effect is called.
* Concurrent: As previously mentioned the parts of the code where a side effect is called are well defined and the language enforce us to separate pure functions from the side effects parts. This makes easier for the compiler determine optimizations in the software. IO is usually a blocking part of a software so a well differentiated code allows the compiler to make the code concurrent and the programmer to have fine-grained control over interactions with IO or keeping state.

Because of the previously listed features we consider that Haskell is worth to use it as the primary language to develop the mappers and reducers, leveraging the development with its features. Giving us fine grained control when it comes to performance while keeping the code declarative about what is making.