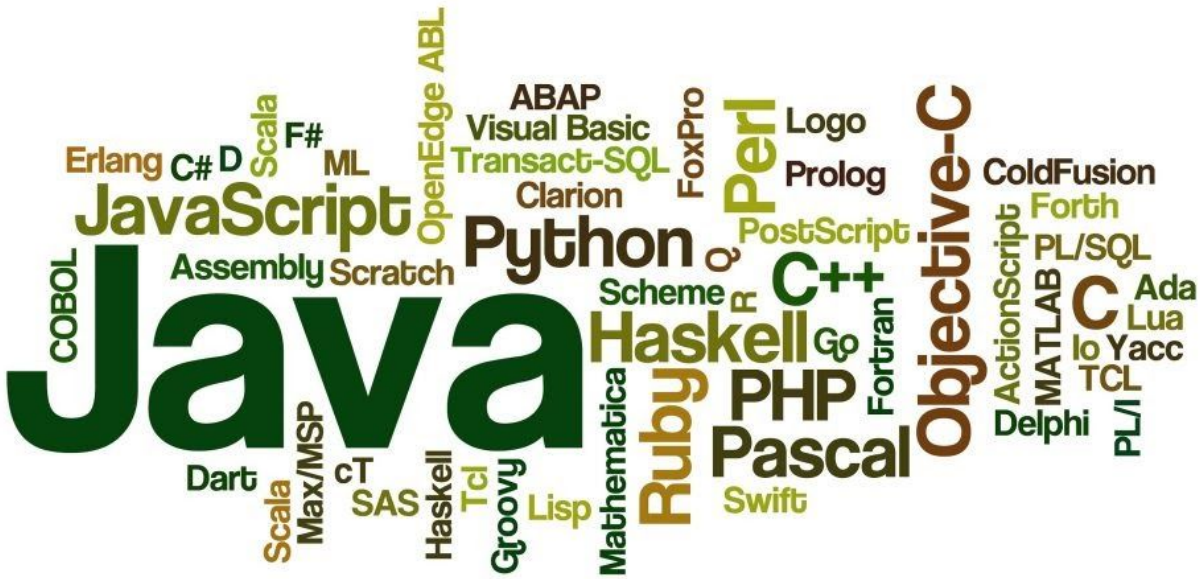
A Comparative Essay on the Conceptual Approaches in Lua and Haskell

Researching and Comparing Lua and Haskell.



Shane Butt

14th January 2020

BSc Computer Science Year 3

INTRODUCTION

Lua

Lua is a multi-paradigm scripting language, encompassing the imperative and functional paradigms, it also incorporates the procedural, instance-based programming (prototypal / prototype programming / classless / etc.) and object-oriented programming paradigms. It was designed in 1993 by the Computer Graphics Technology Group (Tecgraf) at the Pontifical Catholic University of Rio de Janeiro.
Lua does not fully subscribe itself to any single paradigm and instead was designed such that it could be expanded upon; for example, Lua is not completely an object-oriented language and does not have “Objects” in which to work with, however, through the use of metatables Lua can implement inheritance. Inheritance is usually achieved through the combination of different features of Lua, most commonly modules and metatables are used together to create “classes” (prototypes) which can be used in other scripts. However, in Haskell, inheritance doesn’t exist and modules only serve the purpose of fulfilling modularity and code reuse, they do not help implement any feature of other paradigms.
Lua’s native table implementation is absurdly powerful, allowing the implementation of classes and namespaces, in addition to this, Lua also has full lexical scoping which makes it able to employ the principle of least privilege (PoLP) - meaning every module must only be able to access the information and resources that are necessary for its purpose.
Furthermore, the full reference interpreter is roughly 250kB in size when compiled. This means Lua is extremely lightweight and makes it far easier to adapt to a varied range of applications.

Haskell

Haskell is a purely functional language and adopts no other paradigms into its design. It was designed by a whole slew of people and was first released in 1990, 30 years ago.
Haskell is a purely functional programming language which means that it has no side effects; this means that generally, functions don’t modify state variables outside of the local environment. In essence, functions usually don't have any observable effects besides returning a value to the operation executing the function. Despite this, Haskell can model the impure functions of other languages with a pure function which returns a side effect which subsequently gets executed; this is done through the use of the construct which exists to represent side effects, among other things - these are called Monads, and serve a vast array of uses from error-handling to software transactional memory which is a counter-form of lock-based synchronisation. Monads will be discussed more in-depth throughout the essay over different topics due to their importance in Haskell. Moreover, in Haskell, explicit type annotations may be omitted in many cases, though not all. The language is capable of reconstructing types as it were for many terms. Indetermination is possible, and in this case, you must, of course, manually annotate types, and the indeterminacy appears as indeterminacy usually does, in the presence of multiple valid determinations or no valid determinations, the former case being ambiguity. Haskell also has type classes. A type class defines an interface of sorts which may be implemented by various types.
For example, many algebraic constructs exist as type classes within Haskell. A monoid type class could require a type to implement a function mempty, which returns the monoidal identity, whose type is the type implementing the type class, and mappend, which is, as you might suspect, the associative binary operation for the monoid.

Concise Analysis of Programming Languages

Modularity & Separation of Concerns

Modularity and Separation of Concerns are design principles aimed at separating programs into their distinct pieces. This is usually done such that each part of the program has its distinct meaning and purpose, it also helps keep code clean and manageable. When code is written in this way, it removes a lot of clutter and junk from classes/files where it may not necessarily be needed; however, some programmers argue that this is "software bloating" and can just lead to unnecessary time-wasting (though negligible in everyday scenarios).

Haskell

Haskell handles modularity in a very standard way using 'modules', this is, in fact, similar to Lua, which also uses modules to separate code into different concerns, handling different problems within and allowing the programmer to effectively 'reuse' code. See ***Code Examples 1, 2 & 3*** for an example of this.
Furthermore, if your code is self-contained and isn't that "loosely coupled" (reliant) with each other you can re-use them wherever needed. This makes writing code much more manageable and maintainable. Writing code in this way also makes your program more robust and able to be upgraded and added on to.

Lua

In Lua, modules have a slightly different role in that they allow for prototype programming (a form of OOP) via the use of metatables and metamethods to create 'objects' (named prototypes) as well as supporting code reuse and code separation. This functionality of Lua's modules allow for some pretty powerful programs, being able to loosely emulate objects and perform some object-oriented-programming principles into the fray can help bring more power to the programmer giving them more choices and access to the resources they otherwise would not (overriding built-in methods with metamethods such as the __call() metamethod which is fired when a (meta)table is called like a function). ***Code Example 4*** is a display of this unique functionality.

Polymorphism

At its core, Polymorphism is a feature of a programming language which allows processes to use variables of different types at different times. There are several different forms of Polymorphism such as parametric, ad-hoc polymorphism, and subtyping polymorphism. The most common of these is subtyping seen most commonly in Java and C++. In Java, polymorphism plays an extremely important role in implementing inheritance and allows objects to have different internal structures under the same external interface. This then allows for some very flexible interfaces, for example, a class called "Vehicle" could have sub-classes of "Car", "Truck", and "Bike" all of which have their own "drive()" method doing different things. If you instantiate an object "vehicle" of type "Vehicle" but invoke the "Car" constructor, the compiler at run-time will know to use the "Car's" "drive()" method and not anything else.

Haskell

Most polymorphism in Haskell falls under two types: ad-hoc polymorphism and parametric polymorphism. Parametric polymorphism refers to a value whose type contains one or more - unconstrained - type variables. Ad-hoc polymorphism refers to a value which can adopt several types because it (or a value it uses) has been given definitions for each separate type.
The following example(s) illustrate parametric polymorphism within Haskell:
`id :: a -> a` : this means that the function can return any value so long as the type of the value given in the argument is of the

same type: `Char -> Char` for example.
`map :: (a -> b) -> [a] -> [b]` : may operate on any function type; so long as each type variable (a, b) are consistent throughout. Multiple appearances of a type variable must take the same type wherever it appears.
Haskell also has a property called parametricity which is a somewhat limiting property but means that these unconstrained type variables behave the same regardless of the type.
An excellent example of Ad-hoc polymorphism would be the addition operator which performs differently depending on what types it acts upon.

Lua

Unlike Haskell, Lua doesn't implement standard polymorphism as seen in Java or C++, this is because of it's dynamic type system which cuts away the explicit polymorphic concepts involving type overloading. An elegant example of a polymorphic function in Lua can be seen below, utilising tuples to achieve this goal. While only a short snippet of code, an example of a "polymorphic" function in Lua can be seen under ***Code Example 5***.
As simple as it is and without any functionality, this method can be considered polymorphic. The method could utilise conditional statements to perform different actions based on the arguments provided within the tuple, another example of this would utilise an array of parameters instead of a tuple.

Input, Output, and Functional Purity

First and foremost, we can see that there are a few approaches to IO, there is also a clear division between how pure functional languages and more imperative and less pure languages handle IO; it can then be argued that this exposes the practical difficulty of the pure paradigm, however, these languages have found reasonable methods of handling IO without compromising their purity beyond reason.

Haskell

Haskell is a purely functional programming language, which is to say that mutability is generally absent and expressions have referential transparency. Purity and the accommodations required for it to function gracefully should not be underestimated as an influence on the rest of the language, and it is a powerful point of comparison. Where an impure language will change, a pure one will force you to recreate, modification is replaced by taking the original and creating something new using it. This has a great many benefits regarding predictability, consistency, and safety of programs, but it also induces a considerable change in the mode of thought when programming. Haskell also does not truly have statements, do-notation may superficially appear to reveal some sort of statements in the language, but this is merely syntactic sugar, though it does reflect quite powerfully how monads may be used to model imperative semantics. Referential transparency is a useful consequence of purity, and it implies that an expression may be replaced by the value it evaluates to without changing the program (only partially, it may change the actual computations done of course, but the point of the matter is that the results are the same); a consequence which follows from referential transparency is that expressions may not have side effects.
However, to echo the common wisdom that no pure principle will survive practical work, purity, taken to its extreme, destabilized by its rigidity, will pass over into its negation. In our case, this manifests quite clearly in the realm of I/O, which presents a particularly intractable difficulty for any pure language. The reason why is clear to see when we consider how referential transparency could hold under output specifically, how can any expression sending the outside world arbitrary information and with the executive power to read arbitrary information in from the world be expected to be referentially transparent? How may any function be expected

to be pure and give us the same results for the same inputs when it may read input from the outside world? The answer, of course, is that it may not be expected to do such a thing, and the solution of the language to this situation is as cunning as it is practical. Monads have been anticipated in this discourse above, regarding imperative semantics, but they will find their true relevance now. Within Haskell, IO is to be done through a monad, which holds a "value" and allows you to run a function using what it contains, however, the function must return (mandated and enforced by types) value within the same monad. In ***Code Example 8*** we may use the line input arbitrarily, in our computations, but we may not take it outside of the monadic context. This solution does, however, introduce additional difficulty for those learning the language, as it is an unusual and uncomfortable thing to deal with compared to an impure language such as Lua.

Lua

As Lua is an imperative language it has no concern for functional purity whatsoever and more often works with side-effects, this makes I/O extremely simple, so much so that it comes with an easy-to-use library (technically it is a table, though for simplicity it will be referred to as a library).
Lua has an incredibly straightforward approach to input and output, it comes with a library solely dedicated to providing access to input and output features, though this library is primarily focused on file manipulation. The IO library that comes with Lua derives from C and has two styles for file manipulation: implicit file descriptors and explicit file descriptors. The former utilises operations to set the default input and output files, thus all IO operations are performed on those files. The latter is naturally the opposite to this, wherein you supply the input and output file explicitly. The library also has three pre-defined descriptors with their standard meanings from C, these are io.stdin, io.stdout, and io.stderr.
File descriptors are a user-data containing the file stream with a distinct metatable created by the IO library. Generally, all IO operations return nil upon failure alongside an error message, upon success an IO operation will return non-nil. Examples of the operations one can perform using the IO library are shown under ***Code Example 6***.
Generally, in imperative languages input and output are modelled as side effects, take the “print(...)” function, for example, it has no return value and this in a purely functional language such as Haskell it has no place, this is, however, where Monads serve one of their purposes.

Language Data Types

Haskell

The primary area of distinction and interest within Haskell are its algebraic data types, which are deemed so because they are formed by an algebra over data constructors, which themselves take data in a specific pattern (similar to how functions take data in a specific pattern, hence they are called constructors, they are similar to constructing functions) and then produce values of given types. The algebra in question is one of sums and products, where sums allow for a type to be constructed via different constructors altogether, and products allow for one constructor to take multiple values of differing types.
In ***Code Example 7*** we witness a function pattern matching on data constructors to access the data within a value of an algebraic data type. The first thing to note is that data types may be recursive, allowing for simple construction of things such as linked lists. The second to note is that they provide the means to create just about whatever data structures you would like. One important thing to consider is that though sum types and inheritance might seem similar and often are used for similar purposes, sum types are something of converse to inheritance as sum types (in Haskell) subsume multiple interfaces under a single type whereas inheritance subsumes multiple types under a single interface.

Lua

Lua's data types, contrary to Haskell, are quite simple. We have the usual rogues gallery which is not entirely worth covering in detail in a relative comparison such as this. Lua is dynamically typed and possesses the types nil, function, boolean, number, string, function, user-data, thread, and table. Most of these are relatively self-explanatory and user-data is unnecessary for our discussion. As you might expect from functions appearance, Lua has first-class functions and, consequently, higher-order functions. As it also has closures, it has access to one form of creating objects, in the form of returning an interface of functions which act on captured variables. The prime data structure in Lua is the table, which is an associative array. It can serve in its virtuous generality as a wide variety of data structures and is serviceable for all you could ask of it. It allows you to set a metatable for it which alters its behaviour, allowing for another form of objects by way of prototypical inheritance.

Reflection

In total, we may conclude that Haskell and Lua occupy primarily distinct use cases. Lua is less novel, more limited, and much less adept at general-purpose abstraction than Haskell, but it is also more lightweight, easier to use, and embeddable, which is a strong factor in its favour. Lua's type system is dynamic and less expressive than Haskell's, but this can be a utility in writing code quickly, where the Haskell type system can be very strict and demanding of correctness. Some would argue this is no advantage for Lua, though it can be said that its ease of using and learning is most definitely an advantage. Both languages are alike in their distinctness from languages focused on OOP, though in all else they seem to differ by quite a bit. Lua comes from a much more familiar mode of thought for most programmers though Haskell in itself is quite familiar to functional programmers. Haskell is a virtuous language in many regards and has clever solutions to difficult problems, but can prove unwieldy and unapproachable for many. The category-theoretic constructs which it is full of are off-putting for the average programmer who has not been exposed to such things. It bears the motto "avoid success at all costs", which is telling and intriguing. It can be interpreted as holding principles and good design above what may practically succeed better in the market of languages as it were. Lua is quite opposed, being purposefully accessible, small, and has sacrificed in some ways for these goals. Both approaches would have a place, anyone reasonable would agree, in a healthy language ecosystem. There will always be a necessity for vanguards and homemakers, those who will chart out the lands we will tomorrow call home, and those who remain grounded in our established civilizations.

References & Code Examples

The following websites helped in structuring and wording my document, as well as paving the foundations of what to discuss within this document. They also played a role in choosing the languages discussed:

- Wikipedia (2019). *Comparison of programming languages (basic instructions)*. [online]. (Last Updated 13th November 2019). Available at: https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions) [Accessed Jan 2020].
- Wikipedia (2019). Comparison of programming languages. [online]. (Last Updated 7th January 2020). Available at: https://en.wikipedia.org/wiki/Comparison_of_programming_languages [Accessed Jan 2020].

The below websites lent a hand in researching and learning about both Lua and Haskell, most information was retrieved from memory and personal experience, however, in lacking areas these websites made up for it:

- lua.org (2020). Lua 5.3 Reference Manual. [online]. (Last Updated 18th June 2018). Available at: https://www.lua.org/manual/5.3/contents.html#contents. [Accessed Jan 2020].
- Wikibooks (2020). Haskell Wikibook. [online]. (Last Updated 15th November 2019). Available at: https://en.wikibooks.org/wiki/Haskell. [Accessed Jan 2020].
- Wikipedia (Year). Lua Programming Language. [online]. (Last Updated 17th January 2020). Available at: https://en.wikipedia.org/wiki/Lua_(programming_language). [Accessed Jan 2020].
- Wikipedia (Year). Haskell Programming Language. [online]. (Last Updated 24th January 2020). Available at: https://en.wikipedia.org/wiki/Haskell_(programming_language). [Accessed Jan 2020].


```haskell
module Lib
    ( main,
      string
    ) where -- module which exposes main and string, which may be defined below the where

import Data.List -- import another module
import qualified Data.Map as Map -- import another module qualified, so that to use members from it you must prefix them with Map. in this case

main = 3
string = "4324"
```

Code Example 1 - Haskell Modules

```lua
local Module = { }

function Module:MethodA(arg1, arg2)
    print("Hello, world!")
end

function Module:MethodB(arg1, arg2)
    print("Oh no! The cookies are gone!")
end

return Module
```

Code Example 2 - Making a Module

```lua
local Module = require("Module")
Module:MethodA()  -->>Hello, world!
Module:MethodB()  -->>Oh no! The cookies are gone!
```

Code Example 3 - Requiring a Module

```lua
local Shape2D = { } -- The "Object"
Shape2D.prototype = { Width = 0, Height = 0 } -- The prototype
Shape2D.metatable = {
    __index = function(t, k) -- called when Shape2D.metatable[key] cannot be indexed
        return Shape2D.prototype[k]
    end
}

function Shape2D.new(obj) -- creates new Shape2D
    obj = obj or { }
    setmetatable(obj, Shape2D.metatable)
    return obj
end
```

Code Example 4 - Metatables

```lua
local tab = { }
function tab:polymorphic(...)
    --> do something
end
```

Code Example 5 - "Polymorphic" Function

```lua
file = io.open(fileName [, mode]) -- opens a file with the specified mode given by the string mode and returns a file descriptor
--[[
    mode can be one of the following:
    "r" read mode (default if no mode supplied);
    "w" write mode;
    "a" append mode;
    "r+" update mode with previous data preserved;
    "w+" update mode with previous data erased;
    "a+" append update mode, previous data preserved and writing only allowed at the end of the file
]]--

io.close([file]) -- file:close() equivalent, without a file supplied will close the default file.
io.flush() -- file:flush() equivalent over default file
io.input([file]) -- with a supplied file name it will open the named file (in text mode), and use it as the default descriptor. when called with a file descriptor it will set that descriptor as the default input file. returns the current default input file without any arguments supplied.
io.lines([filename]) -- opens the given file in read mode and returns an iterator function which returns a new line from the file each time it is called, you can therefore do:
for line in io.lines(filename) do ... end

io.output([file]) -- similar to io.input() except it operates over the default output file.
io.read(format1, ...) -- io.input():read() equivalent
```

Code Example 6 - Lua IO Operations

```haskell
data Prod = Cons Int String -- takes an int and a string

x :: Prod
x = Cons 3 "hi"

data Sum = One Prod | Two String

y :: Sum
y = One x
z :: Sum
z = Two "hi"

f :: Sum -> String
f (One (Cons num str)) = str
f (Two str) = str
```

Code Example 7 - Pattern Matching Data Constructors