A case study on Haskell
Nish (Paradox?)

---

1. Introduction

Haskell [1] is a functional programming language [2] [3] that was first developed in the late 1980s. It is named after the logician Haskell Curry [7] and is known for its strong type system [8], lazy evaluation, and purity. Haskell is a language that is becoming increasingly popular in the programming world, due to its expressive power and ability to write code that is concise, clear, and maintainable. Here, we explore a few paradigms that make Haskell a functional language. We explore ideas such as Pattern Matching, Guard Clauses, Higher-order functions, Currying, and Lambda Expressions. Finally, we also compare Haskell with imperative languages.

2. Haskell's General Properties

2.1. Statically Typed

Haskell is a statically typed language, which means that every expression has a type that is determined at compile time. Haskell uses type inference to determine the type of an expression, so in many cases, you do not need to specify explicitly the type of a variable or function.

2.2. Lazy Evaluation

One of the main features of Haskell is its support for lazy evaluation. In Haskell, expressions are only evaluated when they are needed, rather than being evaluated eagerly as in many other programming languages. This means that if a function does not use a particular argument, that argument will not be evaluated. This can lead to more efficient and concise code, as it allows you to work with infinite data structures and avoid unnecessary computations.

2.3. Pureness

One of the main features of Haskell is its support for lazy evaluation. In Haskell, expressions are only evaluated when they are needed, rather than being evaluated eagerly as in many other programming languages. This can lead to more efficient and concise code, as it allows you to work with infinite data structures and avoid unnecessary computations. One of the properties that make Haskell a purely functional language is that it is referentially transparent. That is: everything is immutable and expressions never have *side effects* [12]. This is also enforced by the *idempotency* nature of these expressions. The same call to the same function with the same arguments provides the same set of results.

3. Haskell Basic Syntaxes and Data Structures

3.1 Syntaxes

Haskell has a clean and concise syntax that is designed to be easy to read and write. In Haskell, indentation is used to denote code blocks, rather than braces or keywords. This means that the structure of the code is visually apparent, making it easier to read and understand. Haskell programs are composed of expressions, which are evaluated to produce values. Expressions are made up of literals, variables, functions, and operators. Haskell is whitespace sensitive, so indentation is used to define code blocks. Comments start with **"--"** and continue to the end of the line.

```
functionName arg1 arg2 = body
```

Haskell syntax is based on mathematical notation, with functions defined using the "=>" symbol and arguments separated by spaces.

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

This function takes two integers as arguments and returns their sum. The first line of the function definition specifies the type signature of the function, which tells us that the function takes two integers as input and returns an integer as output.

3.2 Basic Types and Data Structures

In Haskell, there are no primitive data types like ***int***, ***char***, or ***bool*** as in many other programming languages. Instead, Haskell has a system of type classes that allows you to define generic operations that work for many different types. However, there are some basic types that are built into Haskell, including:

1. **`Int`**: fixed-precision integers, typically 32 or 64 bits depending on the platform
2. **`Integer`**: arbitrary-precision integers, with no practical limit on size
3. **`Float`**: single-precision floating-point numbers
4. **Double**: double-precision floating-point numbers
5. **Char**: Unicode characters, enclosed in single quotes, e.g., 'a'
6. **Bool**: boolean values, either True or False
7. **()**: the unit type, which has only one value, also written as ().

These types are instances of various type classes, such as **`Num`** for numeric types, **`Eq`** for equality comparison, and **`Show`** for converting values to strings.

Haskell has a rich set of built-in data structures, including:

- **Lists**: ordered collections of values of the same type, created using square brackets and commas, e.g., **`[1, 2, 3]`**

- **Tuples**: ordered collections of values of different types, created using parentheses and commas, e.g., `(1, "hello")`
- **Functions**: first-class citizens that can be passed as arguments, returned as results, and composed together
- **Algebraic data types**: custom data types created using the data keyword, e.g.,

```
data Color = Red | Green | Blue
data Maybe a = Just a | Nothing
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Haskell also supports type classes, which are like interfaces that define a set of functions that a type must implement. Some common type classes include:

- **Eq**: types that can be compared for equality, using the == and /= operators
- **Ord**: types that have an ordering relation, using the <, >, <=, and >= operators
- **Show**: types that can be converted to strings, using the show function
- **Read**: types that can be converted from strings, using the read function
- **Num**: types that support numeric operations, such as addition, subtraction, multiplication, and division
- **Monad**: types that support sequential computation, using the >>= and >> operators. [9]

4. Functions and Pattern-Matching

4.1 Functions

In Haskell, a function is a named sequence of expressions that takes zero or more arguments and returns a value. Functions are defined using the following syntax:

```
functionName argument1 argument2 ... = expression
```

For example: if we want to have a function that takes in 3 *int* arguments and returns the product of those 3 numbers as an int, we can declare it as

```
prodThreeArg :: Int -> Int -> Int -> Int
prodThreeArg a b c = a * b * c
```

One thing to note here is that it's not necessary to use ( ) braces to do a function call in Haskell like in other languages because function calls are treated like any expressions in Haskell. However, we can provide associativity using ( ) braces to represent what gets evaluated first during the evaluation. For example for a 2-argument function "f", called `f (x+1) y`, the `(x+1)` gets evaluated whose value is then used for evaluating the function call.

4.2 Pattern matching

Pattern-matching is a powerful mechanism in Haskell that allows us to match values against patterns and extract data from them. Pattern matching can be used in function definitions, data type definitions, and even case expressions. The basic syntax for pattern-matching in function definitions is:

```
functionName pattern1 = expression1
functionName pattern2 = expression2
...
```

Each pattern is matched against the function's arguments in turn, and the corresponding expression is evaluated if a match is found. For example, the following code-block represents a function that returns a true boolean value if it pattern-matches with an even number

```
isEven :: Int -> Bool
isEven n = n `mod` 2 == 0
```

Common pattern-matching mechanism in Haskell is through recursion. Eg:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial(n-1)
```

4.2.1. Matching with lists

The lists are defined recursively using the `:` (cons) operator and the empty list `[]`. We can pattern match against lists using the `:` operator to extract the head and tail of the list. For example in the following code, the _ represents an item in the cons operator followed by a sub list that follows the item. This is one of the common patterns in Haskell while doing pattern-matching in terms of list.

```
lenOfList :: [Int] -> Int
lenOfList [] = 0
lenOfList (_ : tail) = 1 + lenOfList tail
```

4.2.2. Matching with Custom Data Types

We can define our own custom data types using the `data` keyword. We can also pattern match against these data types using the same syntax as for tuples and lists. For example:

```
data Person = Person { name :: String, age :: Int }
getAge :: Person -> Int
getAge (Person _ age) = age
```

4.3. Guard Clauses

Haskell's idiomatic use for ***if*** expressions and pattern-matching are actually achieved through guard clauses which are expressions that consist of a series of boolean expressions followed by an arrow and corresponding expressions. The basic syntax looks like

```
function x
       | boolean_expression_1 = expression_1
       | boolean_expression_2 = expression_2
       | …
       | otherwise = expression_n
```

The pipe symbol '|' denotes the guard clause as shown in the syntax above. Guard clauses help in avoiding nested if-else statements and make the code more readable (and thus idiomatic). For example, the following code redefines factorial computation using guard clauses.

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | otherwise = n * factorial (n - 1)
```

5. Higher-order Functions

In Haskell, functions are first-class citizens, which means that functions can be passed around like any other arguments/values. This allows for the creation of higher-order functions [2] [3] [4], which are functions that take other functions as arguments or return functions as their results. This allows for a great deal of flexibility and power in programming, as it allows us to abstract away common patterns and create new functions that are more expressive and concise.

Some common examples of higher-order functions in Haskell include

5.1. map

This function takes a function and a list, and applies the function to each element of the list. For example, the following code will double each element of the list `[1, 2, 3]`:

```
map (* 2) [1, 2, 3] -- Outputs: [2, 4, 6]
```

The type definition of *map* is: `` `(a -> b) -> [a] -> [b]` ``. That is: ***map*** is a function that takes two arguments: The first argument is a function (a -> b), which takes an element of type **a** and returns an element of type **b**. The second argument is a list **[a]** of elements of type **a**. The result of applying ***map*** to these two arguments is a new list **[b]** where each element is the result of applying the function (a -> b) to the corresponding element in the input list **[a]**.

5.2. filter

This function takes a function and a list, and returns a new list containing only the elements of the original list that satisfy the function/expression. For example, the following code will return a list containing only the even numbers from the list `[1, 2, 3, 4]`:

```
filter even [1, 2, 3, 4] -- Outputs: [2, 4]
```

The type definition of *filter* is: `(a -> Bool) -> [a] -> [a]`. That is: The first argument is a function (a -> Bool) that takes an element of type **a** and returns a **Bool** value indicating whether the element should be included in the output list. The second argument is a list **[a]** of elements of type **a**. ***filter*** applies the given function to each element in the input list and includes only those elements for which the function returns True. The result is a new list **[a]** of elements that satisfy the given condition.

5.3. folding

This is a powerful mechanism to reduce a list of some types into a final single value, generally used for aggregation. There are two folding mechanisms [1]: *foldl* (fold-left) and *foldr* (fold-right). *foldl* applies a function to a list from left to right, while *foldr* applies the function from right to left. The function takes a function, an initial value, and a list, and applies the function to the initial value and each element of the list, from left to right. For example, the following code will sum the elements of the list [1, 2, 3]:

```
foldl (+) 0 [1, 2, 3] -- Outputs: 6
```

***Note**: Here, in Haskell, the basic operators like +, -, >, < are also treated as function and can be passed to other functions for further operations. This is also one of the general patterns in Haskell that make its functional paradigm more powerful.*

5.4. List Comprehension

List comprehensions are a concise way of generating lists in Haskell. They allow you to specify the elements of a list based on one or more input lists and a predicate which is a pre-condition for the generation. For example, the following code block generates a list of even numbers:

```
evenNumbers = [x | x <- [1..10], even x]
```

The general syntax of a list comprehension is as follows: `[expression | generator, generator, ..., predicate, predicate, ...]`.The *expression* is the value to be included in the resulting list, and the *generator* is a way to generate values for the expression. The *predicate* is a condition that must be true for a value to be included in the resulting list.

6. Function Composition, Lambda Expressions, and Currying

6.1. Function Composition

This is an important concept in Haskell and is a key part of the functional programming paradigm. It is based on the idea of function composition in mathematics. The result of the composition is a function that takes one or more arguments and applies each of the component functions to those arguments in turn. In Haskell, function composition is denoted by the dot operator (.).
For example, here the function *f* increments an integer input and the function *g* doubles the integer input. So, using the composition operator (dot), we can combine the two functions to

create a new composite function *h*. The evaluation happens from the rightmost function towards the left. So, *h* behaves mathematically as `(x+1)*2`.

```
f :: Int -> Int
f x = x + 1

g :: Int -> Int
g x = x * 2

h = g . f
res = h 5 -- result is: 12
```

The function composition in Haskell is defined using the following type signature:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

The . (dot) operator takes two functions as arguments: the first function takes a b as input and produces a c as output, and the second function takes an a as input and produces a b as output. The resulting composed function takes an a as input and produces a c as output.

6.2. Lambda expressions

In Haskell, anonymous functions are known as lambda expressions. They allow us to create small, one-off functions without having to give them a name and can be plugged into any compatible expression.

A lambda expression is defined using the \ symbol, followed by a list of arguments separated by spaces or commas, and then a body expression separated by a -> symbol. For example, the expression `(\x -> x + 1)` is a lambda. We can use this directly to evaluate an input value:

```
result = (\x -> x + 1) 2 -- result is 3
```

One of the powerful and common paradigms for lambda is to use them for higher-order function patterns (map/filter/fold). Example:

```
vals :: [Integer] = [-2, -1..10]

-- simple filter
print( filter (\x -> x>3) vals ) -- gives >3 integers

-- function composition through map-filter-fold pattern
foobar :: [Integer] -> Integer
foobar = sum . map (\x -> 7*x + 2) . filter (>0)
print(foobar vals) -- Result is: 405
```

6.3. Currying

This is a technique in functional programming where a function that takes multiple arguments is transformed into a series of functions that each take a single argument. In Haskell, all functions are curried by default. That is: every function takes one argument and returns another function that takes the next argument. So, even if the user declares a function that “seems” to take multiple arguments, is actually curried.

Currying allows us to partially apply functions, which means passing fewer arguments than the function expects. When we partially apply a curried function, we get back a new function that takes the remaining arguments.

For example, we can partially apply the `add` function to create a new function that adds `3` to its argument: `addOne = add 1`

The resulting `addOne` function takes an Int as input and adds 3 to it. We can call `addOne` with an argument like this: `result = addOne 5 -- result is 6`

7. Example programs

Following are a few sample programs in Haskell to demonstrate some of the basic constructs:

7.1. Extract all the digits

```
toDigits :: Integer -> [Integer]
toDigits n
       | n>0 = toDigits (n `div` 10) ++ [n `mod` 10]
       | otherwise = []
```

This program extracts all the digits from an input integer in the form of a list of integer. The program demonstrates the usage of recursion as well as guard clauses. ***[6]***

7.2. Check Prime

```
-- pattern matcher
isPrime :: Integer -> Bool
isPrime 0 = False
isPrime 1 = False
isPrime 2 = True
isPrime 3 = True

isPrime n
       | n `mod` 2 == 0 = False
       | n `mod` 2 /=0 = checkDivisor n

-- For an input number 'n', we check if it has an odd divisor
-- iterating only till √n
checkDivisor :: Integer -> Bool
checkDivisor n = or [(n `mod` x /= 0) | x <- [3,5..(ceiling (sqrt (fromIntegral
n)) + 1)] ]
```

In this program, we have primitive pattern-matcher that tests for input ints 0, 1, 2 and 3. Then there are guarded clauses where it (i) checks for even numbers (even no. > 2 are non-prime) (ii) Checks for odd divisors. The *checkDivisor* returns True if any odd divisor is found for the input number. This function also uses basic list comprehension to generate the list of odd numbers to check for divisibility.

8. Contrasting Haskell with imperative languages

The most notable difference between functional and imperative programming is the way in which they express computation. Functional programming focuses on the use of pure functions and immutable data structures, which leads to more declarative and concise code that is often easier to reason about, while imperative programming focuses on changing state and mutating data, which can lead to more complex code that is harder to reason about.

8.1. Pure Functions

In Haskell, functions are pure, meaning that they have no side effects and always return the same output for a given input. In contrast, Python and C allow functions to have side effects, such as modifying global variables or printing output.
For example, consider the following Haskell function takes two integers and returns their sum. It has no side effects and will always return the same output for the same inputs.:

```
add :: Int -> Int -> Int
add x y = x + y
```

In Python, the same function might look like the following function, which has the side effect of printing to the console, and could also modify global variables, depending on the scope of those variables.:

```
def add(x, y):
	print("Adding:", x, y)
	return x + y
```

8.2. Immutable Data Structures

In imperative languages, variables are generally mutable. That is: their values can change throughout the program execution. In Haskell, variables are immutable by default. This makes it easier to reason about the code, as you can be sure that the state of the program will not change unexpectedly. In contrast, Python and C allow mutable data structures, which can make it harder to reason about the code.
For example, in the following code block, **list1** is an immutable list, which is then applied to the map function to create a new list **list2** where each element is twice the corresponding element in list1. Since **list1** is immutable, we can be sure that it will not be modified by the map function.:

```
list1 = [1, 2, 3]
list2 = map (*2) list1
```

8.3. Lazy Evaluation

Haskell uses lazy evaluation, meaning that expressions are only evaluated when they are needed. This can lead to more efficient code, as it avoids unnecessary computation. On the other hand, imperative languages such as Python and C use strict evaluation, meaning that expressions are evaluated immediately.
For example, the Haskell following program creates an infinite list [1, 2, 3, ...] and then takes the first 5 elements of the list. Since Haskell uses lazy evaluation, the infinite list is never fully

created, and only the first 5 elements are computed: `take 5 [1..]`. Contrastingly, in imperative languages, you'd have to create an infinite loop whilst generating the list or have to store a pre-computed list.

8.4. Higher-order Functions

Higher-order functions are an important feature of functional programming and are not commonly found in imperative languages. As mentioned in the previous sections about higher-order functions, in Haskell we treat functions as first-class citizens and can be passed as arguments to other functions, returned as results from functions, and assigned to variables. the following code defines a higher-order function that takes a function and applies it twice to an input:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

In an imperative language like JavaScript, the following code defines a function that takes a function and applies it twice to an input: `function twice(f, x) { return f(f(x)); }`

9. Conclusion

Concludingly, in this term paper/case study, we explore the different functional paradigms of Haskell such as its purely functional nature, conciseness & expressiveness, expression evaluation, higher-order functions, etc. Finally, I also studied different contrasting factors that make Haskell different from imperative languages. However, because of the complexity of various topics like monads [9] and type systems [8], I wasn't able to go in-depth into such topics. In all, Haskell has definitely helped me to explore different programming paradigms and I can foresee using it in some real applications [10] [11].

10. References

1. "HaskellWiki." 17 Apr. 2023, wiki.haskell.org/Haskell.
2. Hudak, Paul and Joseph H. Fasel. "A gentle introduction to Haskell." SIGPLAN Not., vol. 27, no. 5, 1 May. 1992, pp. 1-52, doi:10.1145/130697.130698.
3. Alkhulaif, Shams A. "A Functional Programming Language with Patterns and Copatterns." Dissertation, Brock University, 2020, dr.library.brocku.ca/handle/10464/14868.
4. "A Gentle Introduction to Haskell, Version 98." 16 Feb. 2019, www.haskell.org/tutorial.
5. "Learn Haskell in Y Minutes." 17 Apr. 2023, learnxinyminutes.com/docs/haskell.
6. "Lecture notes and assignments." 4 Apr. 2013, www.cis.upenn.edu/~cis1940/spring13/lectures.html.
7. Contributors to Wikimedia projects. "Haskell Curry - Wikipedia." 23 Jan. 2023, en.wikipedia.org/w/index.php?title=Haskell_Curry&oldid=1135193516.
8. "Making sense of the Haskell type system by Ryan Lemmer at FnConf17." YouTube, 17 Apr. 2023, www.youtube.com/watch?v=tJNU1H9XewM&themeRefresh=1.
9. "All About Monads - HaskellWiki." 2 May. 2022, wiki.haskell.org/All_About_Monads.
10. Dreimanis, Gints. "11 Companies That Use Haskell in Production." Serokell Software Development Company, 3 May. 2022, serokell.io/blog/top-software-written-in-haskell.
11. "home." xmonad - the tiling window manager that rocks, 3 Apr. 2023, xmonad.org.
12. "Side effect (computer science) - Wikipedia." 1 Mar. 2023, en.wikipedia.org/w/index.php?title=Side_effect_(computer_science)&oldid=1142347657.