

HASKELL-BASED PROGRAMMING LANGUAGE ANALYSIS REPORT

Yuxuan Hou 4962273

January 2022

ABSTRACT. As a statically typed pure functional programming language, Haskell has type inference and lazy evaluation. As one of the many programming languages on the market today, Haskell is not as popular as other well-known languages such as Python. But normalized pure functional programming has brought some loyal fans to Haskell. This report on the analysis of Haskell programming language will start with a basic introduction of Haskell, including its origin and its programming features. After this, this report will try to analysis how the higher-order function works and its related issues. At the end, this report also compare the strengths and weaknesses of Haskell and analyze them.

Keywords: Haskell, Functional Programming

1 Introduction

In this section, this report focuses on the origin and development of Haskell. It also gives a brief introduction to the style of writing Haskell.

1.1 Haskell's History

Proofs are programs and propositions are types. In the 1930s, the American mathematician Alonzo Church introduced the Lambda Calculus, a system for defining functional calculus by using symbolic expressions for the binding and substitution of variables, which is an important cornerstone of functional programming languages. 1958, John McCarthy, a professor at Stanford University, was influenced by a language called IPL (Information Processing Language) developed at Carnegie Mellon University, developed a functional programming language called Lisp. Although IPL is not strictly a functional programming language, it already has some basic ideas, such as higher-order functions. After the birth of Lisp, more and more people started to join the functional programming camp.

In the 1960s, Peter Landin and Christopher Strachey at Oxford University made it clear that λ -arithmetic was of great importance to functional programming, and in 1966 developed a functional programming language called ISWIM (If you See What I Mean), a purely functional language based on λ -arithmetic programming language, which to some extent laid the foundation for functional programming language design.

In recent decades, many theories based on functional programming have emerged from the research of scholars in various universities. In 1987, at the Functional Programming and Computer Architecture Conference (FPCA) in Oregon, USA, participants decided to design a functional programming language that was open-source, free, simple in syntax, stable, and easy to use. It was named Haskell after the American mathematician Haskell Brooks Curry, in honor of his work in λ -arithmetic and combinatory logic. Haskell Curry gave the design of functional programming languages a very solid theoretical foundation. Song (2014)

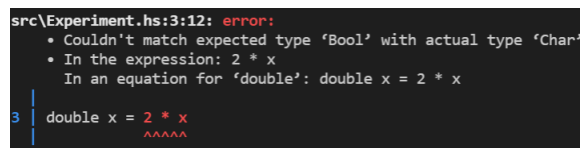
1.2 Haskell's Programming Style

As a programming language that has been discussed by many bright academics with PhDs, it may not look very friendly to some people. It is not written in a traditional programming

language style. It is also not as simple as Python, with a straightforward language that is easy to understand; and with the support of several third-party libraries, Python become one of the most used programming languages in interdisciplinary subjects today.

- 1). Haskell, being purely functional programming, is destined for a niche audience. Unlike imperative programming, where we give the program a series of commands to run, in functional programming our code actually consists of expressions to be evaluated, which then also means that each expression should have its specific type. In other words, Haskell doesn't require the writer to tell it what it needs to do, it requires you to tell it what it is, and to tell it what you want the output to be. So, Haskell does not allow the editor to infer the behavior of the program and facilitates the programmer to better detect the correctness of the results. Because of this, a nice Haskell should also be very simple, because you don't need too many loops or nesting back and forth to tell the compiler what the program is doing at this step.
- 2). Haskell is also a very lazy language. Unlike strict languages, a strict language makes the function arguments fully evaluated before they are passed to the function. As the name implies, sometimes strict languages specify the order in which function arguments are evaluated. But Haskell, being a lazy language, is the exact opposite. When Haskell evaluates a function argument, the process of evaluating the value of that argument is delayed as long as possible by lazy Haskell until it must be used and cannot be lazy any longer. This has two main benefits, the first being that when code changes are needed, very little code modification is required to reach the goal; and the other benefit is that lazy evaluation somehow extends the data structure to infinite length.
- 3). Furthermore, Haskell is statically typed. It has type inference. It's like a little kid learning math, as long as $a = 1 + 2$, it doesn't care if you tell it that a is an int type, it just uses what it learned about addition to compute the math problem and then fills in the answer to a . Lipovaca (2011)

However, the static nature of Haskell also makes it more "rigid" for writers to write. Because each expression in Haskell has a type determined at compile time. All types combined by function applications must match. If they don't, the program will be rejected by the compiler. Website (2022) For example, if we specify that when the input is Char and the output is the result of a judgment on this input char (Bool), then if our actual output is still char, the system will report an error showing that the actual output does not match the specified output, as shown below.

A screenshot of a Haskell compiler error message. The text is as follows:

```
src\Experiment.hs:3:12: error:
• Couldn't match expected type 'Bool' with actual type 'Char'
• In the expression: 2 * x
  In an equation for 'double': double x = 2 * x
3 | double x = 2 * x
  |             ^^^^^
```

Figure 1: If the type is not match

2 Higher-order functions and related closures

2.1 Higher-order functions

As functional programming, Haskell makes flexible use of the higher-order function, which means that it treats functions as parameters and return values to be passed. This form is also nested, as in $\text{fun } x \rightarrow (\text{fun } y \rightarrow y + x)$. This case is often used when multiple functions are nested, especially when using Haskell's various bindings are added, such as *where*, *let*, or even *pattern*

matching. When multiple functions are stacked together, the variables are substituted for each other in an iterative process. In the same way as in the usual problem solving procedure, the basic problem description gives the value of a base variable. We answer the question to find another value for the formula corresponding to this variable value, and then we put this resulting value in the formula we are now calculating, and the resulting value is then applied to the next formula at However, imagine that in the next question of the problem title, a special case is set up in which the value of the base variable is redefined to a new value. In this case, we can tell: yes, I should use the new value of the variable in this environment (question). But imagine if this happens in a program? Since two variables have the same variable name, but in different environments, it is easy for our rigid program to execute the wrong variable value. For example:

```
Let f =
  let mess = 10 in x -> x + mess
in
  let mess = 20 in f 20
-return 30
```

Unfortunately, the program does not know whether the mess is supposed to be 10 or 20. Or, in fact, the programmer does not explicitly specify the value. More explicitly, the program is not told which environment the mess is supposed to use.

2.2 Closures

The use of higher-order functions is very clever and largely streamlines the code. But before using them we must make auxiliary declarations of variables, so that the program can specify which function variable corresponds to which value in multiple nested functions.

Still using the above example, the reason for the problem is that the program doesn't know which value in the range the variable actually corresponds to. Then, since this is the case, we can declare the value corresponding to this variable in a combinatorial way. That is, using the environment (variable, value) as an aid. But this method of using the environment to modify the evaluated object does not solve the nest-nest function. e.g., *(fun x -> fun y -> y+x) 1 evaluates to fun y -> y+x*, which results in the inability to bind to x when we apply this function. So we need to replace the free variables in this function using the environment that defines the nested function. This substitution binding will iterate over the expressions, assign the value to the variable, and then evaluate the expressions... obviously, this is a very inefficient process.

However, if we specify a substitution, instead of replacing the nested function when it arrives at a function (which will be referred to as a lambda for convenience), we can use a package that combines the environment and the lambda to complete the substitution when the nested function is applied. This package is closure.

Essentially, we can understand closure in this way: it is basically an pair of environment plus function pointers. With closure, we go from a substitution-based syntactic to a closure-based syntactic process.

So how do we compile closure in Haskell? The closure conversion process is: first we represent the value of the function as a function pointer plus an environment pair; then we need to make all functions take the environment as an additional parameter, and then use the environment as a point to access the parameter. Chong (2018)

But still, variables that are defined outside of the nested function are still required. However, during Haskell functional compilation, variables that appear in the nested function will not appear, because they will appear as closures. These two cases also mean that we don't need to compile all variables with closures, because that would make the whole program very inefficient. We only need to compile for those variables that appear in the nested function, or for those variables that are within the function but whose corresponding function value is escaped from its definition. Gabriele Keller (2021)

2.3 Some Thoughts about Closures and Objects

In many ways, it seems that both closure and object accomplish the same thing, which is to encapsulate data or functionality in a single logical unit. For example, if we create a student class in Python and then instantiate it as an object, the student object contains a lot of built-in data and functionality. When the internal function methods of the student class are called, the resulting data will be stored in a variable stored as an object property, which means that the result is still within the internal scope of the class. This situation leads to an interesting statement: Closures are poor man's objects.

But while it may be that the process is the same, given the differences in programming languages, personally both approaches are tools to achieve the end. For that matter, missingfaktor missingfaktor (2012) points out that for object-oriented programming languages, as exemplified by Java, true lexical closures are not supported at the language level. To reach the ultimate goal, Java engineers use anonymous inner classes to include scoped variables. Haskell, being a functional programming language, like mentioned at the beginning, its statements are composed of expressions to be evaluated. So it has no language-level support for real objects.

So for the relationship between closure and object, I think we can change the metaphor: a hammer is a poor man's corkscrew, and a corkscrew is a poor man's hammer. In other words, most things have multipurpose, regardless of whether it is convenient or not.

3 Haskell's Strengths and Weaknesses

There are two sides to everything. I think that for Haskell, the strengths of the language come from its purely functional programming, its laziness, and its static characteristics. However, it is also the reason why many programmers find Haskell not very user-friendly. In this section, I have selected the top three pros and cons from Haskell mentioned in the Stackshare website for analysis. stackshare (2021) This section will also analyze the situations and areas in which Haskell is suitable for use.

3.1 Strengths

3.1.1 Purely-functional programming

Among all the user comments about the advantages of Haskell, functional programming is the one that comes up most often. First of all, as a member of the functional programming language, it is natural that it has the advantages and features of functional programming. This is reflected in the fact that functional programming makes code more clean. For example, the handling of nested functions, the nice clean inlining and the use of syntax bindings reduce the amount of code related to looping, declaring variables, or iterator design compared to imperative programming. This further gives Haskell's functional programming the edge in complex examples. In addition, the support for lazy evaluation, as described in the introduction to the Haskell language features, has the advantage of avoiding unneeded operations, thus improving performance and making it possible to create infinite data structures. Bhadwal (2021)

3.1.2 Statically typed and type-safety

Haskell's statically type is also one of the features of this programming language. Its static feature provides static security. This is the biggest advantage of statics. Because in static types, the compiler can check if each function that is written has been written with the correct name and whether the parameters of the correct type are provided. This way, many errors can be found and located at compile time. This also allows the code to have relatively good performance. Gros-Dubois (2017)

3.1.3 Open source

Haskell is not a new language, on the contrary, it has been around for quite some time and its long development has led to the accumulation of a large number of libraries. According to the official Haskell website, there are now 6954 freely packages available, such as xml, stm, time, parsec, etc. Website (2022)

3.2 Weaknesses

3.2.1 Too much distraction in language extensions

As with many other programming languages, Haskell takes a different approach to language design; GHC allows its core language to extend on a per-project or per-module basis by changing the semantics of the language. Diehl (2020) Such extensions do increase the flexibility of the language to some extent, but some of the language extensions, even if officially specified, are immature. Inappropriate language extensions not only cause problems when compiling, but also sometimes limit the range of Haskell compilers that can be used. In addition, many extensions are too complex compared to Haskell 98 and result in very difficult to understand error messages. Wiki (2021)

3.2.2 Error messages can be very confusing

Unlike the straightforward detail of compiler error messages when using Python, Haskell's system error messages are very strange, except of course for most of the checks and hints for statics, etc. (which indicate that you are missing variable values or that the output of a function does not match the definition, etc.) Haskell's error messages seem too unintuitive and useless, especially if the user is not familiar with the language. Not only that, but these confusing errors will show up even worse and more confusing in more advanced functions. The vicious circle of errors caused by a lack of understanding of Haskell, but confusing prompts that confuse the user even more about writing programs, become one of the reasons why many users dislike Haskell.

3.2.3 Libraries have poor documentation

Perhaps it is the first two problems that put most users out of the gate, and the difficulty for novices to get started that has led to fewer people using Haskell. This has also led to fewer libraries and library documents being shared by users. Many users complained that some definitions in the library documents provided by Haskell were obscure, and the lack of vivid examples made users spend a lot of time and experience to study how to use them. Mob_Of_One said in a reddit discussion community *"Types are valuable, but examples are critical for anything non-trivial. 90% of the times I've been unable to do something in a jiffy in Haskell. The other 10% was just slow-down due to needing to perform type-tetris"*. Mob_Of_One (2014) The unreadable user guide documentation has certainly rejected a large number of potential users again, making it one of Haskell's weaknesses.

3.3 Haskell's Application

Haskell's most famous application is solving Facebook's spam mail problem. Many people think that its application exists only in academic research, or learning to study and understand functional programming. However, due to its static as well as functional characteristics, as seen in the advantages of Haskell presented in the previous section. If Haskell is applied to a project, it will greatly reduce the time and effort of post-maintenance. Not limited to development, Haskell can also be used in artificial intelligence algorithms due to higher-order functions and laziness, as John Hughes wrote in his article on how to implement alpha-beta algorithms using functional

programming. Hughes (1989) In addition, we can see more use of Haskell from its official website report. Maruseac (2018)

4 Conclusions

In this report on the Haskell, some basic Haskell features are introduced, such as functional programming and its static characteristics. The problems associated with higher-order functions and how they can be solved by using closure are also analyzed step by step. Besides, a summary of the strengths and weaknesses of Haskell is presented. To summarize, Haskell is a programming language with a very clear two-tier review. Those who like it praise its static features and the elegance and beauty of its purely functional code, as well as the simplicity of its implementation; however, its writing style and highly logical use of the language make many people give up. Nevertheless, there is no denying that Haskell is a very good example of functional programming that can be understood and learned. This has caused more and more people to take it seriously, and it is increasingly used in a variety of fields.

References

- Bhadwal, A. (2021). Functional programming: Concepts, advantages, disadvantages, and applications.
- Chong, S. (2018). Cs153: Compilers lecture 12: Closures and environments.
- Diehl, S. (2020). What i wish i knew when learning haskell.
- Gabriele Keller, L. O.-D. (2021). Concepts of program design supplementary lecture notes abstract machine.
- Gros-Dubois, J. (2017). Statically typed vs dynamically typed languages.
- Hughes, J. (1989). Why functional programming matters. *Computer Journal*, 22:98–107.
- Lipovaca, M. (2011). *Learn You a Haskell*. No Starch Press.
- Maruseac, M. (2018). Haskell communities and activities report.
- missingfaktor (2012). "closures are poor man's objects and vice versa" - what does this mean?
- Mob_Of_One (2014). There is a documentation problem, stop pretending there isn't. an extraction of a twitter thread.
- Song, Z. (2014). *Introduction to Haskell Functional Programming*. Posts & Telecom Press.
- stackshare (2021). Haskell vs java vs python.
- Website (2022). Haskell official website.
- Wiki (2021). Haskell.