CPSC 353 Report

Scott Fitzpatrick
Chapman University

November 29, 2021

Abstract

The Haskell portion of this report is a kickstart guide for programmers with experience in imperative languages to learn functional programming, and more specifically, Haskell. From preparation of learning Haskell to the completion of a small Haskell program, this guide will be going over many of the basics in a follow-along fashion. There will be multiple small tasks which make up the entirety of the guide. It is meant to be unique in the aspect that it focuses on not only the general differences between imperative and functional programming, but also notes the syntax differences to avoid any confusion and frustration. Therefore, this guide is suitable for any programmers who are accustomed to programming in imperative languages and find functional programming to be less intuitive.

Contents

1 Introduction

First and foremost, this report will cover the programming language: Haskell. As stated in the abstract, Haskell will be examined in a kickstart guide format for programmers that are used to imperative programming languages.

2 Haskell

2.1 Preparation for Haskell

To learn and become familiar with Haskell, it is best that you prime yourself for success. I will share my opinions on the best ways to introduce yourself to Haskell.

For installation of Haskell, see this page this page. It provides adequate information and links for installation. If lucky, this process should not take long. Keep in mind that if you are using Windows, it may be optimal to use Windows Subsystem for Linux (WSL) or Docker.

It is also advantageous to use a good IDE that supports Haskell through plugins. I recommend Visual Studio Code with the Haskell Syntax Highlighting plugin. It colorizes the code which makes viewing easier in all aspects. The syntax will be easier to get used to if you can consistently associate different colors with variables, functions, etc. I started without one of these plugins, and when the font is all white, things get confusing very quickly. Haskell Runner as another extension for Visual Studio Code that allows you to run haskell files within the IDE. However, I personally found that a classic terminal window works fine. You may also use other IDEs such as Emacs, Atom, or IntelliJ.

Here is a trap to be weary of: assuming Haskell will be similar to other programming languages and that the process will feel familiar. This naive and costly mindset led me and some of my peers to fall behind after hitting unexpected and unfamiliar roadblocks. Be prepared to not only notice the differences in syntax, but rethink the functionality of code down to the core. Functional programming is much more theoretical and logic based. We will cover the details of this later. For now, it is best to prime yourself by preparing for completely new concepts that take mental flexibility to wrap your mind around. So be prepared to learn Haskell without any predispositions since things will seem somewhat different to imperative languages. It is also very important to pay close attention to the differences from imperative programming languages such as Python, Java, or C. For this tutorial, we will be comparing Haskell code with C.

2.2 Introduction

My assumption is that you already know how to program in imperative languages such as C++, Python, or Java. Understanding how to code in these languages will make learning Haskell easier. However, it is important to make sure to prepare for the differences of functional programming.

Functional Programming differences to prepare yourself for:

1. One of these topics is the use of “pure” functions. These are bare-bone mathematical functions that your programs will function off of. Do not think of pure functions as instructions. Instead, think of them as definitions. It may be helpful to think of pure functions in Haskell similar to how you would think of actual mathematical functions such as those taught in school. Furthermore, pure functions do not have “side effects”. This means that evaluating an expression will not change some internal state which would later cause that same exact expression to have a different result. In a pure function, the same expression can be evaluated an infinite number of times and as long as the arguments are the same, the return value will stay the same.

2. Immutable data is another aspect of functional programming. Similar to declaring a variable as a constant in an imperative language, Haskell variables will always be constant and you should not expect to mutate them. Though this may seem like a limitation, especially to an imperative programmer, immutables in Haskell have some advantages. Immutability in functional programming is essential for adopting a mathematical approach. This is because, in math, objects never change their state, and therefore requiring this immutability in programming will inadvertently make the code more mathematical. Immutable data can also be a major advantage when creating multithreaded applications. The consistency and immutability of variables is extremely useful for multithreaded applications because it prevents conflicts with different threads since variables do not change and the threads will use standard data. Of course, this aspect would be more relevant at an advanced level but it is something to keep in mind.

3. Functional programming languages are declarative instead of imperative. Imperative programming uses a sequence of instructions on how to complete a task to obtain the result. On the other hand, in declarative programming the result is declared as the outcome of function applications. View this example:

```haskell
-- HASKELL
sum [] = 0
sum (x:xs) = x + sum xs
```

```c
// C CODE
int sum(int *arr, int arrSize) {
  int i;
  int sum = 0;

  for (i = 0; i < arrSize; i++)
    sum += arr[i];

  return sum;
}
```

   Notice how in the imperative C code, there is explicit initialization of the variables and an explicit loop that will modify the data. These are explicit instructions for how to calculate the sum. Therefore it is imperative. On the contrary, the Haskell code, which is declarative, simply declares how the result is found with the use of addition and recursion. The second line of code is basically a mathematical function which contrasts with the C code, which is more like a series of instructions.

4. Another key difference of functional programming that imperative programmers may not be used to is lazy evaluation. With lazy evaluation, expressions only get evaluated if they need to be. With imperative languages which are considered "strict", an entire function can run, even if some of the computation is excessive and a waste of time and energy. Some functional programming languages such as Haskell will only compute what is needed to solve the task at hand. This is because evaluation is completely deferred until results are needed by other computations. This is obviously an advantage in the aspect that it can be much faster and efficient in certain scenarios. Also, since Haskell has these "non-strict" semantics, it allows for the processing of formally infinite data.

2.3 Declecaring Variables with Types

Now that we have primed ourselves for learning Haskell, installed the proper software, and configured our IDE, we will begin with the basics such as types, functions, and more.Similar to imperative languages, when declaring variables in Haskell we must associate a type so that the computer knows what it is dealing with.

```haskell
-- HASKELL
x :: Integer
x = 1

y :: Bool
y = true

pi :: Float
pi = 3.1415
```

```c
// C CODE
int x;
x = 1;

bool y;
y = true;

float pi;
pi = 3.1415;
```

At this point, the differences between Haskell and C are not too drastic, but be sure to note the differences in syntax and remember that the Haskell variables are immutable unlike the C variables.

2.4 Defining a Function

To define a function follow the example:

```haskell
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range min max val = val < min || val > max
```

```c
// C CODE
bool out_of_range(int min, int max, int val) {
  if (val < min || x > max)
    return true;
  else
    return false;
}
```

Similar to declaration variables with types, we must do the same with functions. The first line first states the name of the function followed by `::` . After this, the function parameter types are listed. In this case there are three integer types as the parameters that represent the min, max, and input value. The last type, which in this case is a boolean, is at the end. This may look confusing considering all of the types next to each other and separated with `->`. It is important to remember that the last type is the return type and there can be as many input types as needed. Also using `::` between a function or variable name and its type can look strange at first but it is generally easy to get used to. In a way, it can be advantageous in the aspect that it stands out which is optimal when defining the type of a new variable.

The second line actually defines how the function obtains the result. The function name followed by the input variable names are located on the left hand side of the equals sign. On the other side, the computation

is made with the assistance of the less than and greater than operands (which are functions themselves). In this case, it simply checks if the value is below the minimum or above the maximum.

Notice how this function declaration, when it comes down to it, is really a boolean value. This contrasts with the C function which is instructions to get this same value. This ties back to the aspect of declarative versus imperative programming.

2.5 Calling a Function

Calling a function is rather simple. View this example:

```haskell
-- HASKELL
result :: Bool
result = out_of_range 4 7 5
```

```c
// C CODE
bool result;
result = out_of_range(4, 7 ,5)
```

The function is simply called by typing the name of the function followed by the inputs. In this particular example, we are assigning the result to a boolean variable for sample context. The result would be false. One syntax difference that may take some getting used to is the fact that there are no parentheses in Haskell like there usually is in imperative languages such as C in this example. I personally always associated functions and calling functions with parentheses, and when working with Haskell it took me a while to get used to seeing a function with its parameters simply listed out behind it with spaces. Furthermore, remember to use the right type as inputs to the function. If, for example, a float was used instead of an integer, it would create a type error.

2.6 Function "let"

With Haskell, you may assign a value to a name, which can later be used to define the last expression or result. This can create more flexibility when programming with Haskell. Consider this example:

```haskell
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range minBound maxBound val =
  let out_of_lower_bound = minBound > val
      out_of_upper_bound = maxBound < val
  in
  out_of_lower_bound || out_of_upper_bound
```

```c
// C CODE
bool out_of_range(int minBound, int maxBound, int val) {
  bool out_of_lower_bound = minBound > val;
  bool out_of_upper_bound = maxBound < val;

  return out_of_lower_bound || out_of_upper_bound;
}
```

Take your time observing the format of the syntax. The Haskell code is very similar to C in this instance. One of the biggest differences in terms of appearance is how the function declaration uses a let and an "in"

statement. With this format, Haskell let functions are still functional programming since the function is still equivalent to a boolean value. Unlike the C code, there were no step-by-step instructions that led to a return type.

2.7 Function "where"

Similar to `let`, `where` allows you to use multiple variables with assigned values to define the function. However, `where` does this after the initial function declaration. View this example:

```haskell
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range minBound maxBound val = out_of_lower_bound || out_of_upper_bound
  where
    out_of_lower_bound = minBound > val
    out_of_upper_bound = maxBound < val
```

In my opinion, `where` functions look much cleaner and readable than `let` functions, but your choice is obviously up to you. Also, the corresponding C function would just be the same as the previous C example for `let`.

2.8 Using "if", "else if", "else", and "then"

In Haskell you may also use `if` and `else if`, and `else` statements. We will use them in the same function for convenience. View the example below:

```haskell
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range minBound maxBound val =
  if minBound > val then True
  else if maxBound < val then True
  else False
```

```c
// C CODE
bool out_of_range(int minBound, int maxBound, int val) {
  if (minBound > val)
    return true;
  else if (maxBound < val)
    return true;
  else
    return false;
}
```

If you saw, the statements are not too different from imperative C. However, they are followed by `then` which precedes the resulting value of the function determined by the boolean statement. Also, keep in mind that in Haskell, `True` and `False` are capitalized unlike C.

Furthermore, this can be simplified syntactically by using guards:

```haskell
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range minBound maxBound val
  | minBound > val then True
```

```
  | maxBound < val then True
  | otherwise False
```

2.9 Recursion and Pattern Matching

In imperative languages it is routine to use loops in order to obtain results and calculations from whatever the computation may be. However, in Haskell loops do not exist. You may wonder, "How am I supposed to accomplish anything then?". The answer is: recursion. If you are unfamiliar with recursion, it may be good to practice it in an imperative language first. Here is an article on writing a recursive C program for Fiibonacci numbers. Remember that functions are recursive when they call themselves.

For our example, we will also be doing a recursive Fibonacci function. We will explore pattern matching in this example too. View the Haskell sample below:

```
-- HASKELL
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

You may be confused by the — symbol. However, it does not have any value and is just a separator. Also, the preceding second and third line: `fib 0 = 1` and `fib 1 = 1` may be confusing. This is where pattern matching comes into play. Pattern matching is like piecewise functions in math. For example, his would be the corresponding mathematical piecewise function for the Fibonacci function:

$$fib(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n \geq 2 \end{cases}$$

When calling the function, the designated body is chosen with comparison of arguments. In this instance if n is 0 or 1, 1 is returned. If n is greater than or equal to 2, Fibonacci expressions are executed and the recursion commences. The first two rows of this piecewise function act as base cases and the same goes for the Haskell code on lines two and three. It may also be easier to compare this piecewise function with a syntactically simplified version of the Haskell Fibonacci function with "—" guards. Here is that version:

```
-- HASKELL
fib :: Integer -> Integer
fib n | n == 0 = 1
      | n == 1 = 1
      | n >= 2 = fib (n-1) + fib (n-2)
```

Also, here is the imperative C code for the same function:

```
// C CODE
int fib(int n) {
  if (n <= 1)
    return n;
  else
    return fib(n - 1) + fib(n - 2);
}
```

3 Programming Languages Theory

3.1 Introduction to String Rewriting

Why Should One Study Abstract Reduction Systems?

String Rewriting, and more specifically, String Reduction Systems (ARS) are significant because they can be instantiated at various levels of abstraction. This can be done in hardware with algorithms, programming languages, and high-level software engineering. ELABORATE. ARS's can be used to demonstrate and define properties such as normal forms, termination, and confluence.

Though the term "abstract reduction systems" may sound fancy or intimidating, it is not too different from what is studied in math at a middle-school level. Just as mathematical algebra equations can be solved in a variety of ways to obtain the same result, ARS's do the same. Equational theory and rewrite systems are almost equivalent. The main difference is that ARS's are usually evaluated from left to right, unlike algebriac equations which can be evaluated in whichever direction.

3.2 Brief Quickstart: What is an Abstract Reduction System (ARS)?

An ARS is a set of data paired with a set of rules. Oftentimes the data is referred to as objects. The set of rules define the binary relationship between objects. The notation appears as: (A, R) where A is the set of data and R is the set of rules. The binary relationship denoted by $\rightarrow$ and is called the reduction relation. ELABORATE ON DIFFERENT KINDS OF REDUCTION

View this simple ARS example:

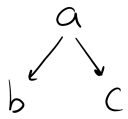


Figure 1: ARS = (A, R)
A = {a, b, c}
R = {(a, b), (a, c)}

We obtain that our set of objects (A) consists of a, b, and c. From (a, b) and (a, c), we also see that our rules are that a $\rightarrow$ b and a $\rightarrow$ c. These rules define the binary relationship between a, b, and c.

Here is another example of a simple ARS where A = {a, b} and R = (a, b), (b, b)}. The visual of this is:

Figure 2: ARS = (A, R)
A = {a, b}
R = {(a, b), (b, b)}

TALK ABOUT THE UNIQUENESS OF THIS ARS AND TRANSITION TO HOW THERE ARE INIFINITE POSSIBILITIES ALL WITH THEIR OWN SPECIAL PROPERTIES AND FEATURES

There are infinite ARS examples that can be viewed for further exploration. For the purpose of keeping this quickstart brief, we will move on and view more specific examples.

3.3 Significant Properties of Abstract Reduction Systems

3.3.1 Termination

From basic intuition you may have guessed that if an ARS is terminating, then it will always stop at some point when executed. If you have guessed this, then you are correct. When there are no loops in an ARS, it will eventually terminate. Here are some examples of terminating and non-terminating ARS's.

Figure 3: ARS = (A, R)
A = {a, b}
R = {(a, b)}

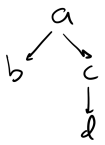


Figure 4: ARS = (A, R)
A = {a, b, c, d}
R = {(a, b), (a, c), (c,d)}

Figure 5: ARS = (A, R)
A = {a}
R = {}

Note how in all of these examples above, the ARS reaches a definite end at some point. Therefore, it is terminating. Here are some examples of non-terminating ARS's. ELABORATE

Notice how in all of these examples there is a loop that prevents the ARS from completely terminating. Even if one path allows for termination, if there are any portions of the ARS where it could loop, the entire ARS is considered non-terminating. When an ARS has a loop, it is non-deterministic, meaning that it will not have the same result every time. ELABORATE

Figure 6: ARS = (A, R)
A = {a, b}
R = {(a, a), (a, b)}

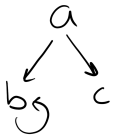


Figure 7: ARS = (A, R)
A = {a, b, c}
R = {(a, b), (b, b), (a, c)}

Figure 8: ARS = (A, R)
A = {a}
R = {a, a}

3.3.2 Unique Normal Form (UNF)

To define a unique normal form, we must first define a normal form. A normal form is an object that no rules can apply to. This also implies that it is the end of the computation and can be recognized when there are no arrows stemming from the object. Here are some examples of normal forms:

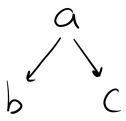


Figure 9: ARS = (A, R)
A = {a, b, c}
R = {(a, b), (a, c)}

A unique normal form (UNF) is the same thing, except it is the one and only normal form of the ARS. It is called the "unique" normal form because it is the only normal form that the ARS will result in. ELABORATE. Here are some examples of unique normal forms:

As you can see, both of these examples reveal that each ARS leads to one UNF. Keep in mind that having

Figure 10: ARS = (A, R)
A = {a, b, c}
R = {(a, a), (a, b), (a, c)}

Figure 11: ARS = (A, R)
A = {a, b}
R = {(a, b)}

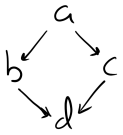


Figure 12: ARS = (A, R)
{a, b, c, d}
R = {(a, b), (a, c), (b, d), (c, d)}

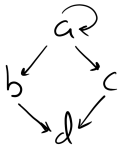


Figure 13: ARS = (A, R)
A = {a, b, c, d}
R = {(a, a), (a, b), (a, c), (b, d), (c, d)}

a UNF does not imply that the ARS will be deterministic, since there can be a UNF in an ARS with loops. The third ARS above is an example of this. CONCLUDE

3.3.3 Confluence

INTRODUCE AND ELABORATE ON CONFLUENCE. A simple way to think of this is that a confluent ARS has a peak for every valley. In the example below, a, b, and c represent the peak and b, c, and d represent the valley.

Furthermore, if the ARS can take multiple routes to achieve the same result, then it is confluent. This can be seen in the confluent example above, since both routes (a $\rightarrow$ b $\rightarrow$ d) and (a $\rightarrow$ c $\rightarrow$ d) lead to the same

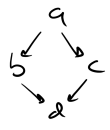


Figure 14: ARS = (A, R)
A = {a, b, c ,d}
R = {(a, b), (a, c), (b, d), (c, d)}

result. Here are more examples of confluent ARS's.

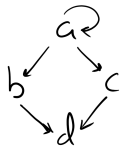


Figure 15: ARS = (A, R)
A = {a, b, c, d}
R = {(a, a), (a, b), (a, c), (b, d), (c, d)}

Figure 16: ARS = (A, R)
A = {a, b, c}
R = {(a, b), (a, c), (b, a), (c, a)}

3.4 Relationships Between the Properties of Abstract Reduction Systems

By now, you may have begun to ponder the relationships between confluence, termination, and unique normal forms. More specifically, you may have suspected that not all combinations of these three properties are possible in ARS's.

Here is a chart with corresponding ARS's to help visualize these relationships. Keep in mind that other ARS's can apply to some of relationship combinations.

Some of the cases that you may have encountered are that if an ARS is confluent and terminating, then it must have a UNF. Similarly, if an ARS is not confluent and then it cannot have a UNF. These relationships can be expressed in mathematical boolean notation:

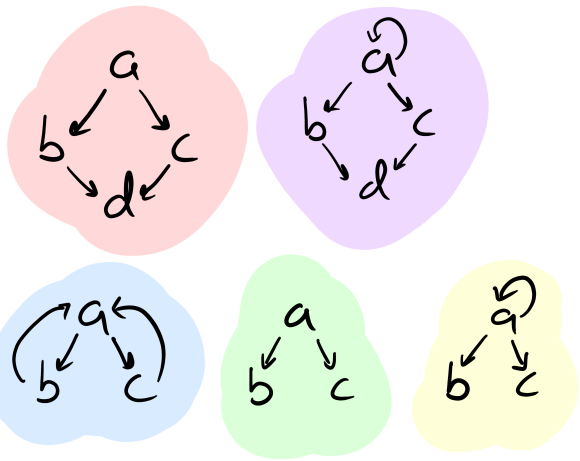
$\neg$ (C $\wedge$ T $\wedge$ $\neg$ UNF) $\neg$ ($\neg$ C $\wedge$ T $\wedge$ UNF) $\neg$ ($\neg$ C $\wedge$ $\neg$ T $\wedge$ UNF)

This can be simplified to:

SIMPLIFIED VERSION

For reference, here is what the boolean notation would look like in C. MAKE SURE THIS IS CORRECT

C	t	UNF
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F




```
!(C && T && !UNF)
!(!C && T && UNF)
!(!C && !T && UNF)
```

This can be simplified too:

```
C && T == UNF
!C && T == !UNF
!C && !T == !UNF
```

3.5 Visualizing Reflexive, Symmetric, and Transitive Traits

For this tutorial we will be using multiple ARS's denoted as (A, R). For each example, set (A) will be written twice as two separate columns. The purpose of this is to help visualize reflexivity, symmetry, and transitivity while providing an additional form of visualization. Comparing these two forms of visualization should make it easier to grasps these concepts. Keep in mind that the color blue will be used to express changes to the visualzations.

3.5.1 Visualizing Reflexivity

Consider this ARS as both a two-column and traditional visualization.

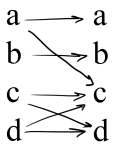


Figure 17: Two-column visualization of (A, R)
A = {a, b, c ,d}
R = {(a, a), (a, c), (b, b), (c, c) (c, d), (d, d), (d, c)}

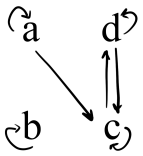


Figure 18: Traditional visualization

When an element points to itself, it is reflexive. In this example, every element is reflexive. Therefore, the entire set A is reflexive. However, if just one element did not point to itself, then the entire A could not be considered reflexive.

3.5.2 Visualizing Symmetry

Taking a look at the same example from Figure 1, notice how the lower portion of the set A appears to be symmetric. This is because not only does c → d, but d → c too. Therefore there is symmetry between these two elements. However, since the entire set A is not symmetric, the set is considered asymmetrical. To make the set symmetric, we would have to make c → a.

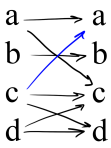


Figure 19: Symmetric two-column visualization of set A

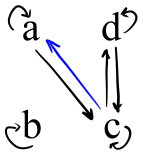


Figure 20: ARS = (A, R)
A = {a, b, c ,d}
R = {(a, a), (a, c), (b, b), (c, c), (c, a), (c, d), (d, d), (d, c)}

Notice how with symmetry, there are arrows going to and from (a, b) and (c, d). This can be rewritten with the proper notation using ↔. The corrected ARS would look like:

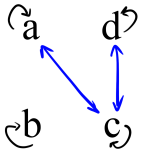


Figure 21: (A, R) with proper notation

With this way of visualizing the set, it is easy to see if it is symmetric as the drawing itself will almost be completely symmetrical. The only asymmetric quality of the drawing is that when an element is reflexive and is pointing to itself, it does not appear to point back. Keep in mind that it is also symmetric because it is simply pointing to itself.

3.5.3 Visualizing Transitivity

$\rightarrow$

INTRODUCE TRANSITIVITY HERE. To simplify the visualization, we will now work with this set:

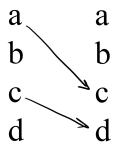


Figure 22: 2 column

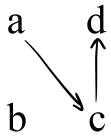


Figure 23: ARS = (A, R)
A = {a, b, c ,d}
R = {(a, c), (c, d)}

Notice how a $\rightarrow$ c and c $\rightarrow$ d. Shouldn't that mean that there should be a shortcut from a $\rightarrow$ d? If this shortcut was added, and R was changed to R = {(a, c), (a, d), (c, d)} then (a, d) would be transitive. The notation for this would be $\xrightarrow{+}$. Here is the visualization of the separated set and the ARS:

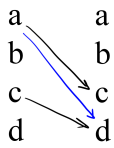


Figure 24: 2 column transitive

4 Project

In this section you will describe a short project. It can either be in Haskell or of a theoretical nature,

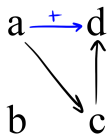


Figure 25: ARS = (A, R)
A = {a, b, c ,d}
R = {(a, c), (a, d), (c, d)}

5 Conclusions

At the beginning of this report, we explored Haskell by working with small examples and differentiating both syntax and theoretical difference between imperative C code and functional Haskell Code. This will be more elaborate when the rest of the report is completed. CONCLUDE THEORY.

References

[PL] Programming Languages 2021, Chapman University, 2021.

[HI] Haskell for Imperative Programmers, Philipp Hagenlocher, 2020.

[HI] Haskell for Imperative Programmers, Philipp Hagenlocher, 2020.

[HR] Haskell/Recursion, WikiBooks, 2020.

[PM] Pattern Matching, Stack Overflow, 2010.

[LE] Lazy Evaluation, Haskell Wiki, 2021.

[ID] Imperative vs. Declarative Programming, Learn To Code Together, 2019.

[HP] Pure, Haskell Wiki, 2021.

[GD] Google Doc, I will cite the sources linked at the end of this doc.