CPSC 353 Report

Scott Fitzpatrick
Chapman University

October 16, 2021

Abstract

The Haskell portion of this report is a tutorial 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 tutorial 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 tutorial. 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 tutorial is suitable for any programmers who are accustomed to programming in imperative languages and find functional programming to be less intuitive.

Contents

1 Introduction

Replace Section 1 with your own short introduction.

1.1 General Remarks

First you need to download and install LaTeX.[1] Alternatively, you can use an online editor such as Overleaf. I prefer to have my own installation, but to get started Overleaf may be easier.

LaTeX is a markup language (as is, for example, HTML). The source code is in a `.tex` file and needs to be compiled for viewing, usually to `.pdf`.

If you want to change the default layout, you need to type commands. For example, `\medskip` inserts a medium vertical space and `\noindent` starts a paragraph without indentation.

Mathematics is typeset between double dollars, for example

$$x + y = y + x.$$

1.2 LaTeX Resources

I start a new subsection, so that you can see how it appears in the table of contents.

- This is how you itemize in LaTeX.
- I think a good way to learn LaTeX is by starting from this template file and build it up step by step. Often stackoverflow will answer your questions. But here are a few resources:
  1. Learn LaTeX in 30 minutes
  2. LaTeX – A document preparation system

1.3 Plagiarism

To avoid plagiarism, make sure that in addition to [PL] you also cite all the external sources you use.

2 Haskell

2.1 Preparation for Haskell

To learn and become familiar with Haskell, it is best that you prepare properly. There are many ways one can set themself up for success. I will share my opinions on the best ways to introduce yourself to Haskell.

For installation of Haskell, see this page https://hackmd.io/@alexhkurz/Hk86XnCzD. 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) LINK or Docker LINK.

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 LINK. 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 LINK 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.

In terms of taking your first steps in Haskell, it is best to understand your own learning style to know which additional sources to focus on. Obviously, it is best to learn from a variety of methods such as reading, watching videos, listening to professors, and trying it yourself. But knowing which method to prioritize is very important. It is always best though to try programming Haskell for yourself, as a true kinesthetic

[1] Links are typeset in blue, but you can change the layout and color of the links if you locate the `hypersetup` command.

experience is a must. You can use such additional sources to assist you in following this tutorial and the small tasks that come with it.

One of the most underrated aspects of preparation for learning anything is a proper mindset. For starting out in Haskell, this is very important. It is easy to make the mistake 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:

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

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. Haskell, for example, 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 Types, Functions, and More

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.

2.3.1 Declecaring Variables with Types

Similar to imperative languages, when declaring variables in Haskell we must associate a type so that the computer knows what it is dealing with.

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

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.

2.3.2 Defining a Function

To define a function follow the example:

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

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 declares 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.

2.3.3 Calling a Function

Calling a function is rather simple. View this example:

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

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.

2.3.4 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: EXAMPLE 4

```
-- 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 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.

2.3.5 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: EXAMPLE 5

```
-- 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".

2.3.6 Using "if", "else if", "else", and "then"

In Haskell you may also use "if", "else if", and "if" statements. We will use them in the same function for convenience. View the example below: EXAMPLE 6

```
-- 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 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: EXAMPLE 6 cont.

```
-- HASKELL
out_of_range :: Integer -> Integer -> Integer -> Bool
out_of_range minBound maxBound val
  | minBound > val then True
  | maxBound < val then True
  | otherwise False
```

2.4 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 https://www.geeksforgeeks.org/c-program-for-fibonacci-numbers/ 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: EXAMPLE 7

```
-- 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: EXAMPLE 7 cont.

```
-- 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: EXAMPLE 7

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

2.5 Kurz Stuff

To typeset Haskell there are several possibilities. For the example below I took the LaTeX code from stackoverflow and the Haskell code from my tutorial.

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

This works well for short snippets of code. For entire programs, it is better to have external links to, for example, Github or Replit (click on the "Run" button and/or the "Code" tab).

3 Programming Languages Theory

In this section you will show what you learned about the theory of programming languages.

4 Project

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

5 Conclusions

Short conclusion.

References

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