CPSC-354 Report

Aidan Wall
Chapman University

November 9, 2021

Abstract

Short introduction to your report ...

Contents

1 Introduction

Replace Section 1 with your own short introduction.

2 Haskell

This section will contain your own introduction to Haskell.

2.1 What is Haskell?

Haskell is "An advanced, purely functional programming language". Haskell is statically typed, meaning that every expression that has a type is determined at compile time. Similar to other languages, type matching and declaration of types are very important, as the code will be rejected at compile time. This is similar to other programs such as C based languages or Java, where if the right types of values are not used it will not compile. Here are some examples of type declarations.

```haskell
-- Basic Type Declarations
char = 'x'    :: Char
int = 22      :: Int
pair = ('a',1) :: (Char, Integer)
```

Haskell also recognizes stings as lists of Chars so to declare strings in Haskell you must make it a list of Chars. Haskell also provides type synonyms which are used for commonly used types.

```haskell
-- String Declarations
type String      =     [Char]
type Restaurant  =     (restName, restAddress)
type restName    =     String
type restAddress =     String
```

Haskell is a "purely functional programming language". Before we get too into depth about the advantages of Haskell we first must understand what a functional programming language is and how it differs from a Imperative Programming Language. A functional programming language is a type of declarative programming that is high-level where the programmer must specify step by step what function the computer will perform. Here is a diagram showing some of the differences between functional and imperative programming styles.

Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

Basically, in a functional programming language, you declare the problem and that is the solution.

"Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions". Haskell makes it very easy to create and use threads. Threads are useful for programmers when a program has many different tasks that are to be done separately from the others. This is useful if one task is blocked, the program can still complete other tasks using different threads and will not be blocked. Overall, threads and simple implementation is a huge benefit to programmers.

Haskell is considered a "Lazy" programming languages, meaning "functions don't evaluate their arguments". This means that it is very easy to write control constructs (such as if/else, AND, OR, NOT) and they can be expressed just by writing normal functions.

```
when p m = if p then m else return()
```

PACKAGES Haskell has a ton of open source contributions and there are many publicly available packages for all sorts of tasks.

2.1.1 Haskell vs Python, a comparison

What are the main differences between Haskell and a language like Python?

@miscWinNT, title = Python vs Haskell, howpublished = https://wiki.python.org/moin/PythonVsHaskell, note = Accessed: 2021-10-15

Python offers programmers many different styles, including procedural, functional, and object-oriented programming styles [**?**] but it is tricky to use and implement functional programming in Python. Haskell is a functional programming language, but can support procedural programming through the use of monads, see more below. Python is a interpreted language. Haskell typically uses a compiler (the one most commonly used that we will learn about is GHC). Haskell and Python are both considered high-level languages, so they compromise on speed compared to C o C++. Through Haskell's type system and declaration Haskell does provide more information at compile time than when running Python, meaning that Haskell does have a speed advantage over Python. "Haskell and Python have strong (not weak) typing, meaning instances of a type cannot be cast into another type" (Same citation). The difference between the two is that Haskell has static typing, and Python has dynamic typing. In statically typed languages, all variables and their types are known when compiled. On the other hand, in dynamically typed languages, the types of variables is not known until run time and "objects can pretend to be different types by providing the correct functions" (Same citation). Some other differences include Haskell having a much greater learning curve for those who are not accustomed to functional programming. To many, lazy evaluation is hard to grasp for someone who has exclusively coded in imperative programming languages.

- List Comprehension Syntax:

  –

- Significant Whitespace:

  –

- Learning Curve:

  –

2.2 More about Haskell

2.2.1 Recursion

Recursion is a very important part of programming in Haskell. Recursion is a very important part of all functional programming languages. A common example of how recursion works is the fibonacci sequence. You write down the mathematical equation and then you are done.

```
Fib b = Fib(n-2) + Fib(n-1)
Fib 4 = Fib(3)  + Fib(2)
      = (Fib(1) + Fib(0)) + Fib(2)
      = (Fib()
```

2.2.2 Haskell for Beginners

- Use recursion instead of loops
  - To calculate the length of a list in Haskell using recursion

```
len[0] = 0
len(x:xs) = 1 + len(xs) -- where x is the head of the list and xs is the tail
```

- No Assignments
  - len[] = 0
  - Write it down as a mathematical equation stating a fact
  - Mathematical equations do not have a direction but Haskell functions do
- Function Declaration
  - When defining a function you initialize them in a similar way to how you call it. First the function name followed by your parameters separated by spaces.
  - You also need to declare the type of what you are working with before you do anything else.
  - For example, if you have a function addI that adds integers, you have to declare it as

```
addI :: II -> II -> II
```

  - This means that in the function addI, you are taking the first II, adding it to the second II and the third value is of type II
- Pattern Matching and Case Distinctions
- Computation is equational reasoning, length function example

```
len(x:xs)= 1 + len xs
len[4,-2,3] = 1+len[-2,3]
len[-2,3] = 1 + (1 + len[3])
len[3] = 1+(1+(1 + len[]))
```

- Using types
  - Declaring return types of functions

```
len:: [x] -> int
```

  - The return type of type list of type int is int

```
Func:: a->b->c->d
```

  - In the function "Func" declared above, (a b and c are input types and last, d is output type). The last variable declared is always the output variable, and the rest are input variables.
  - In a C o C++ an addition function of two numbers would look something like this

```
int add(int x, int y){
return x + y;
}
```

- In comparison, in Haskell, a function to add two numbers would look something like this

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

- It is/should not declared as add(Int, Int) -¿ Int to specify two Int input variables and returns and Int. It is possible to do that syntax and it won't fail but there are many benefits to using arrow notation.
- For example, if we want to implement a function that takes an Int and adds 3 to it. We can use the already declared add function and it would look something like this.

```
plusThree = add 3 :: Int -> Int
```

- This function takes one Int input, then calls add and passes 3 to that the addition function as long and returns an Int

- Correct placement of spaces and parenthesis
- Make use of error messages
  - In C or C++ you would call the function add(2,3) to pass arguments and get the answer 5.
  - In Haskell you would call it add 2 3. Haskell is very strict on syntax so trying anything else would fail. If you tried add(2 3) would fail and the error message would be "maybe you have not applied the function to enough arguments". Everything inside the parenthesis is interpreted as one argument so the function is only registering one argument.
  - Even if you try add(2,3) It will say "Couldn't match expected type Int with actual type (Int,Int)" This is because the expected type is Int -¿ Int -¿ Int, but the actual type of the first argument is (Int, Int). This is Haskell doing doing type inference.
    - 2::Int 3::Int
    - (2,3) :: (Int,Int)
- Function Declaration
- Let Function
  - You can use the keyword "let" in the GHCI compiler to define a variable.
  - For example, let x = 4 is the same as defining x = 4 in python.

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
```

2.3 Lambda Calculus

2.3.1 Intro to Lambda Calculus

Lambda calculus is a mathematical concept that forms the fundamental bases for all functional programming languages. It is a type of notation for functions and applications. It is the basis for applying a function to different arguments. Lambda calculus has three programming constructs :

- Abstraction:
  Abstraction is a concept in lambda calculus to show that that program(also referred to as an expression) and do not depend on the specific variables. In the following example:

  $$\lambda x.e$$

  the program (or function), which has as a formal parameter x. This is called abstraction, because the program

  $$\lambda x.e$$

  does not depend on x anymore, x is abstracted away. More can be seen at [PL]

- Application:

  If e1 and e2 are programs or functions then
  e1e2
  is the program which applies the function e1 is being applied to the argument e2

- Variables:
  The basic functions are just the variables

Lambda calculus provides a much simpler framework and notation for functions.

2.3.2 Comparing Lambda Calculus to C/C++

```
int plus_two(int x) {
return x+2;
}
```

There are a couple differences when implementing in C vs in Haskell.
In Haskell you do not return anything, but you declare the types beforehand, so there is no need for a return statement.
With functions having no names this would be written as (x)x+2, implemented in lambda calculus as

$$\lambda x.x + 2$$

The function x, is having some action done on it (x+2).

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.

[COMP] Functional vs Imperative Programming, 2021

[HASKELL] HASKELL Documentation, 2021