CPSC-354 Report

Anthony Walujono
Chapman University

Haskell is a purely functional programming language. The features of Haskell are:

1. Functional : Haskell codes are expressed in the form of functions.
2. Pure :
    - Haskell expressions have no “side effects”.
    - Every variable or data structure is immutable.
    - Calling the same function with the same arguments will give the same results every time.
3. Statically typed: Every expression has a type and is known at compile time.
4. Type inference: Haskell can infer the type of functions and expressions if we don't explicitly declare their types.
5. Lazy evaluation : Haskell expressions are not evaluated until their results are needed. This enables us to define an infinite list and the compiler will only allocate the ones we use on the infinite list.

The difference between Imperative programming languages (such as C, C++, Java) and functional programming language:

- Imperative programming languages use a sequence of tasks and execute them. While executing the tasks, variables can change state. For instance, a variable x is set 0, then later it can be set to another value.
- Functional programming language performs computation with the concept of mathematical functions that use conditional expressions and recursion.

Here are examples of the difference codes written in Imperative programming language and Haskell:

Python program to display the Fibonacci sequence up to n-th number.

```
nnums  = int(input(“How many numbers? “))

# first two numbers
n1,  n2 = 0, 1
count = 0

# check if the n-th numbers is a positive number
if nnums <= 0;
   print(“Please enter a positive integer”)
# if there is only one number, return n1
elif nnums == 1
   print(“Fibonacci sequence upto”,nnums,”:”)
   print(n1)
# generate fibonacci sequence
else:
   print(“Fibonacci Sequence: “)
   while count < nnums;
       print(n1)
       nth = n1 + n2
       # update values
       n1 = n2
       n2 = nth
       count +=1
```

Here is Haskell code to generate the Fibonacci Sequence :

```
fib = 0 : 1: [a + b | (a, b) <- zip fib (tail fib)]
```

We can see the obvious difference from the above example codes, that Haskell code is very efficient and short, it only take 1 line. Compare to Phyton codes take several lines.

Haskell function syntax:

Function name should start with lowercase.

To declare a function: Function name followed by a space then the parameters.
For example: this function name doubleNumber will double a given number

```
doubleNumber x = x +  x
```

To declare a variable type use double colon punctuation ::
For example:

```
x :: Int
x = 5
```

Variable x is declared with type Int and has value 5.

There is no assignment in Haskell, = is not an assignment, instead = is a definition. In this example x defined to be 5.

Haskell variables are not mutable, in this example x cannot be changed later. For example:

```
x :: Int
x = 5
x = 10
print x
```

The line x = 5 will results in an error of multiple declaration of 'x'

Defining a function type:

```
functionname  :: type -> type
sumOfNum :: Integer -> Integer
```

This defines the function sumOfNum which takes an Integer as input and yields Integer as output.

Haskell uses Pattern Matching to evaluate a function.
Example of pattern matching:

```
numToStr :: Integer -> String
numToStr 1 = “One”
numToStr x = “Error, not a number”
```

When we call numToStr, the patterns will be checked to make sure it conforms to a defined pattern. In this example only number 1 matches the pattern, if not, it will default to x.

Haskell data Types:

- Int : the size depends on the machine) min value -2^63 and max 2^63 for 64 bit machine.
- Integer : the value is unbounded.
- Float : is a real floating point with single precision.
- Double : is a real floating point with double precision.
- Bool : is a boolean type value of True or False
- Char: represents a character, denoted by single quotes.
- String : represents strings of character, denoted by double quotes.

Algebraic Data Type: is a user defined data type, created using algebraic operations. The algebra operations are “sum” and “product”.
“sum” means alternation (A | B) meaning A or B but not both.
“product” means combination (A B) meaning A and B together.
Example:

```
data numpair = I Int | D  Double
```

This is just one number, either Int I or Double D

```
data numpair = N Int Double
```

This is a pair of numbers, Int and Double.

Recursion: in Haskell there is no loop, so it uses recursion.
Recursion is a way of defining functions in which the function is applied inside its own definition. In Mathematics definitions are often given recursively, for example, the Fibonacci sequence is defined recursively.

Here are some example of recursive functions:

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

If we call maximum' [3,6,1] , this function will returns 6.

Another recursive function: a quick sort function:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a> x]
    in smallerSorted ++ [x] ++ biggerSorted
```

The main algorithm of quick sort: a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted.

Haskell Lazy evaluation: evaluation of function arguments is delayed as long as possible, they are not evaluated until it actually becomes necessary to do so. An unevaluated expression (called a "thunk") is packaged without doing any actual work.
For example:

```
somefunction 7 (30^3456)
```

When evaluating somefunction the second argument will be packaged up into a thunk without doing any actual computation, and somefunction will be called immediately. Since somefunction never uses its second argument, the thunk will be thrown away by garbage collector.

Another example of Haskell Lazy evaluation :

When we define an infinite list:
infinitList = [10, 20 . .]

This list will only be created up to what you specified and when you need it, although it can generate an infinite list.

Another example of Lazy evaluation on infinite list:

evenUpto20 = takeWhile (<=20) [2, 4 ..]

This evenUpto20 function will take even values from an infinite list, but only up to 20.

Subsection Plagiarism:

List of resources:

- School Of Haskell
- Learning Haskell in One Video
- Learnyouahaskell.com
- Learning Haskell