CPSC-354 Report

Anthony Walujono
Chapman University

December 21, 2021

Abstract

Short introduction to your report ...

Contents

1 What is Haskell?

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.

1.1 Haskell Lazy Evaluation

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 ($30^3$456)

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.

The following is an example of Haskell lazy evaluation:
Consider the following Haskell program:

```
Main = do
       let myTuple = ("first", map (*2) [1,2,3,4])
       print "Hello"
       print $ fst myTupple
       print head snd myTupple
```

```
    print (length (snd myTupple))
```

When the first print statement ("Hello") is executed, the expression myTuple is actually still unevaluated, eventhough it is defined before the print statement. It is represented in memory by what is called a "thunk". The program knows how to evaluate this thunk when the time comes, but at that moment, there is no value in myTuple. When it prints the first element of the tuple, it still does not completely evaluate myTuple. When the compiler sees the call to fst function on myTuple it knows myTuple must be a tuple. Instead of seeing a single thunk at this point, the compiler sees myTuple as a tuple containing two unevaluated thunks.

Next, the first element of myTuple is printed. Printing a expression forces it to be completely evaluated. So after this, the compiler sees myTuple as a tuple containing a string in its first position and an unevaluated thunk in its second position.

At the next step, it prints the head of the second element of myTuple. This tells the compiler this second element must be a non-empty list. If it were empty, the program would actually crash here. This forces the evaluation of the first element(2). However, the remainder of the list remains an unevaluated thunk.

Next, it prints the length of the list. Haskell will do enough work here to determine how many elements are in the list, but it will not actually evaluate any more items. The list is now an evaluated first element, and 3 unevaluated thunks. Finally, it prints the full list. This evaluates the list in its entirety. If it did not do the last step, the final 3 elements would never be evaluated.

Most programming languages use an evaluation paradigm called: eager evaluation, which means the moment the program execution reaches an expression, the expression is evaluated. This eager evaluation is used in imperative programming languages to execute the commands in order, it evaluates each expression immediately.

1.2 Advantages of Haskell's Lazy Evaluation

There are several advantages that shows Haskell is Lazy. The advantages include:

- Any code you don't absolutely need is never computed.
- It can define and use interesting structures such as infinite list.

Consider the following example:

```
function1 : : Int
function1 = function2 exp1 exp2 exp3
   where
      exp1 = reallyLongFunction 1234
      exp2 = reallyLongFunction 5678
      exp3 = reallyLongFunction 9876
function2 : : Int -> Int -> Int -> Int
function2 exp1 exp2 exp3 = if exp1 < 1000
   then exp2
   else if exp1 < 2000
      then exp3
      else exp1
```

Comparable code in C, C++ or Java would need to make all the three calls to reallyLongFunction before calling function2 with the results. But in Haskell, the program will not call reallyLongFunction until it absolutely needs to.

So in this example, the program will always evaluate exp1 in order to determine the result of the if statement in function2. However, if exp1 greater than or equal to 2000, then it will never evaluate exp2 or exp3! We don't need them! We'll save ourselves from having to make the expensive calls to reallyLongFunction. As a result, this Haskell program will run faster.

Another example that Haskell Laziness will run faster:

Consider a quicksort function algorithm:

1. Pick a partition element.
2. Move all elements smaller than the partition into one list.
3. Move all elements larger than the partition into another list.
4. Sort these two smaller list recursively.
5. Combine them with the partition.

If we want to just get the smallest 3 elements from the list, In Haskell, this function can easily be implemented:

```
smallest3 : : [Int] -> [Int]
smallest3 input = take 3 (quicksort input)

quicksort : : Ord a => [a] -> [a]
```

Since Haskell is lazy, it will never have to touch the larger half of the partition, even in the recursive calls. In an eager evaluation language (imperative programming languages), the compiler would run the full sorting algorithm. Thus, this would take much longer than we need.

Another advantage of Haskell lazy evaluation is the possibility of creating infinite lists. Here are two ways to create infinite lists:

```
allTwos : : [Int]
allTwos = repeat 2

first4 : : [Int]
First4 = cycle [1, 2, 3, 4]
```

The repeat function will produces an infinite list of 2s.

The cycle function will take a list, in this example is [1,2,3,4] and cycles it into an infinite list.

Infinite lists can provide elegant solutions to many list problems, for example we can take the first n elements of an infinite list and the infinite remainders will stay unevaluated.

1.3 Disadvantage of Haskell's Lazy Evaluation

When using recursion or recursive data structures, unevaluated thunks can build up in heap memory. If they consume all your memory, your program will crash. In particular, the fold1 function suffers from this problem. The following usage of fold1 function will probably fail due to a memory leak.

```
fold1 (+) 0 [1..10^7]
```

Lazy evaluation can also make it harder to reason about the code. For example, just because a number of expressions are defined in the same where clause does not mean that they will be evaluated at the same time

and this can easily confuse some beginner of Haskell programmers.

Lazy evaluation has performance drawbacks as it incurs a significant overhead of keeping non-evaluated expressions around, they can use up storage space and they are slower to work with than simple values.

There are several disadvantages that shows that Haskell is Lazy. The disadvantages include:

- It's much harder to reason about the code's performance.
- You can build up larger collections of unevaluated functions that would be much cheaper to store as value.

Here is an example that shows Haskell code to show that Haskell is lazy:

```
main :: IO ()
main = do
   args <- getArgs
   let file name = head args
   file <- openFile filename ReadMode
   input <- hGetContents file
   hClose file
   let summary = (countsText . getCounts) input
   appendFile "stats.dat" (mconcat
   [fileName, " ", summary, "\n"])
   putStrLn summary
```

The code above shows the problem with closing a file before we use it when using lazy evaluation.

Since hGetContents is lazy, the value stored in the input is not used until it is needed. At this point, you can think of input as a substitute for "hGetContents file". In terms of lazy evaluation, hClose has nothing to wait for, and it executes immediately. At this point in the program, the file is closed, but the input has not been evaluated yet. When you define the summary, you are using input, but you still do not need to evaluate it. The input will be evaluated only when summary is evaluated. Finally, you call appendFile which, similar to hClose, has a function. At this point, summary is evaluated and because of the input as well. But now the file is closed, and the OS would not let you read from it anymore.

1.4 Differences with Other Languages

The difference between Imperative programming languages (such as C, C++, Java, Python) 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 different code written in Imperative programming language and Haskell:

- Python program to display the Fibonacci sequence up the 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 cout < nnums:
   print(n1)
   nth = n1 + n2
   #update values
   n1 = n2
   n2 = nth
   cout += 1
```

- C++ Code for Fibonacci Sequence up to the n-th number:

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

int main() {
    int n, t1 = 0, t2 = 1, nextTerm = 0;

    cout << "Enter the number of terms: ";
    cin >> n;

    cout << "Fibonacci Series: ";

    for (int i = 1; i <= n; ++i) {
        // Prints the first two terms.
        if(i == 1) {
            cout << t1 << ", ";
            continue;
        }
        if(i == 2) {
            cout << t2 << ", ";
            continue;
        }
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;

        cout << nextTerm << ", ";
    }
    return 0;
}
```

- The Haskell code to generate 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. However, Python codes take several lines to create a Fibonacci Sequence. C++ also requires several code to create a Fibonacci Sequence and about the same amount with Python code. However, Haskell uses much less code than C++.

1.5 Haskell Syntax

Haskell function syntax:

1. The Function name should start with lowercase.
2. To declare a function: Function name followed by a space then the parameters.

   For example:

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

   This function name doubleNumber will double a given number.

3. 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.

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

   For example:

   ```
   >> let a = [1,2,3]
   >> reverse a
   [3,2,1]
   >> a
   [1,2,3]
   ```

   The value of a did not change. Calling reverse seem to produce the output we expected, but when we want to see the value of a, a did not change at all. So this is immutable.

5. Defining a function type on Haskell:

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

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

1.6 Pattern Matching

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.

1.7 Recursion

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).

2 Data Types

2.1 Haskell Data Types

Haskell Data Types include:

- Int : the size depends on the machine) min value -$2^6$3 and max $2^6$3 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.

2.2 Algebraic Data Types

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.

3 Parsing Arithmetic Expression

3.1 Ambiguous Grammar

Ambiguous Grammar: if there exists more than one rightmost derivation or more than one leftmost derivation or even more than one parse tree for the given input string.

Example of Ambiguous grammar:

```
Exp -> Exp - Exp
Exp -> Exp * Exp
Exp -> Int
```

Parsing this arithmetic expression 10 - 1 * 5

```
     Exp
   /     \
 Exp  -   Exp
  |      /    \
 Int    Exp * Exp
         |      |
        Int    Int
  10  -  1    *  5 = 5
```

Or another possible parsing tree

```
       Exp
     /     \
```

```
   Exp  *   Exp
  /    \     |
Exp - Exp  Int
 |     |
Int   Int
10  -  1  *  5 =  45
```

Because of ambiguous grammar, there are two different results of parsing the above arithmetic expression 10 -1 * 5 = 5 or 10 -1 * 5 = 45.

Ambiguity in grammar is not good for compiler construction because no method can automatically detect and remove ambiguity. To fix this ambiguous grammar we can add a precedence level to the grammar.

3.2 Non-Ambiguous Grammar

Example of Non-Ambigious Grammar:

```
Exp -> Exp - Exp1
Exp1 -> Exp1 * Exp2
Exp1 -> Exp2
Exp2 -> Int
```

Example of Non-Ambigious Tree:

```
          Exp
        /     \
       Exp  -   Exp1
         |         |
        Exp1      Exp2
       /    \      |
    Exp1 * Exp2    Int
    |         |     10
    Exp2    Int
     |       5
     Int
      1
            1 * 5 - 10 = -5
```

The result of parsing the arithmetic expression 10 - 1* 5 is - 5 using the non-ambiguous grammar.

4 Lambda Calculus

Lambda (represented by \symbol in this report) Calculus is the smallest and simplest programming language. Lambda Calculus consists of a single transformation rule (variable substitution) and a single function definition scheme.

The syntax of Lambda Calculus consists of 3 programming constructs:

1. Abstraction : is function definition.
   Example:

```
\x.e  where e: is an expression
                x : is a variable
        \  : is lambda symbol
```

```
\x.e is a function or a program
```

2. Application:
   Example:

```
e1 e2     e1 and e2 are programs
e1 e2 applies function e1 to argument e2
```

3. Variables: the basic programs are just variables.

An expression is defined recursively:

```
<expression> := <name> | <function> | <application>
<function>    := \ <name>. <expression>
<application> := <expression> <expression>
```

An expression can be surrounded with parenthesis for clarity.
The only keywords used in Lambda Calculus are and . (dot).
The adopted convention rule is that function application associates from the left, the expression is evaluated applying the expressions as follows:

(...((e1 e2)e3)...en)

A Lambda expression is a single identifier.
For example :

```
\x.x
   This expression defines the identity function with a single identity x.
```

Functions can be applied to expressions.
An example of an application :

```
(\x.x)y
   This is the identity function x applied to y.
```

4.1 Rules for dropping Parenthesis

- Abstract syntax: is syntax for trees.
- Concrete syntax: is syntax for strings.

When a tree is linearized into strings, we get many parentheses, so we need rules for dropping parenthesis.

- Application associates to the left.
  For example:

```
xyz  in all parentheses : ((xy)z)
```

- Abstraction is a unary operation, so we can drop parenthesis without creating ambiguities.
  For example:

```
\x.\y.\z.a in all parentheses : (\x.(\y.(\z.a)))
```

- Application has higher precedence than abstraction.
  For example:

```
\x.\y.ab  or \x.\y.(ab) or in all parentheses: (\x.(\y.(ab)))
```

Here are some examples of Lambda Calculus terms:

```
\x.(3+x)
(\x.(6 + x)) 5
(\f.(f 2)) (\x.(x+1))
```

Exercise: Put in the parentheses in the right places in the following lambda expressions:

```
((a b)c) (\ x . x)  =  (abc) \x.x
(a b) (\ x . (a x))   = (ab) \x. (ax)
(\ a . \ b. \ c.) a b c x = \a.\b.\c.(abcx)
(\ x. (y b)) (\ x . (a x)) = \x.(yb)\x.(ax)
(\ x . y) (\ x . y (\ x . a x)) = \x.( (y) \x. (y (\x. (a x))))
((a b) c )\ x. \y . ((a b) c) (\ x . x (a b c)) (\ y . (a b c)) = ( (abc) \x.\y.(abc) \x.x
    (abc)\y.(abc))
```

4.2 Semantics of Lambda Calculus

The Semantics of Lambda Calculus are:

- Reduce a term to another. If it can't be reduced further, the term is called Normal form.
  Example of reduction:

```
(\x.( 1+x)) 5 -> (1+5) -> 6
(\f.(f 3)) (\x.(x+1)) -> ((\x.(x + 1)) 3) -> (3 + 1) -> 4
```

- Substitution: of terms for variables.
  Example:

```
((\x.e1)e2 -> (e2 e1)
```

  Replace every occurrence of variable x in e1 with e2.
  Example of substitution:

```
       (\f.(\x. f (f x)) (\x.x+1)) 3
  =   (\x. (\x.x+1) ((\x.x+1) x)) 3
  =   (\x.x+1) ((\x.x+1) 3)
  =   (\x.x+1) (3+1)
  =   3+1+ 1
  =   5
```

  Substitution can go wrong.

A WARNING Example of wrong substitution:

```
(\f. (\x. f (f x)) (\y.y+x)
  = \x. (\y.y+x) ((\y.y+x) x)
  = \x. (\y.y+x) (x+x)
```

```
        = \x. (x+x+x). <--- This is wrong! The x is captured!
```

We can fix this captured problem by Rename Variables.

- We can rename bound variables.
  Example of bound variable:

  ```
       \x.(x+y)   x is bound variable
                     \  is the binder.
  ```

  Example of Free variable:

  ```
        \x.(x+y) z  z is a free variable.
  ```

- Bound variables are just "placeholders"
- In the above example, x is not special, we can rename with
- This is an Alpha (α) equivalent:

  ```
        \x.(x+y) = (Alpha) \z.(z+y)
  ```

  Here is how to fix the captured variable from the above WARNING Example:

  ```
          (\f. (\x. f (f x)) (\y.y+x)
        = (Alpha) (\f. (\z. f (f z)) (\y.y+x)
        = (S Sharp) \z. (\y.y+x) ((\y.y+x) z)
        = (S Sharp) \z. (\y.y+x) (z+x)
        = (S Sharp) \z. z+x+x
  ```

Exercise of Church Encoding using substitution and renaming:

```
        (\mult. \two. \three. mult two three )
(\m. \n. \f. \x. m (n f) x)
(\f. \x. f (f x))
(\f. \x. f (f (f x)))
```

Solution:

```
   (\m. \n. \f. \x. m (n f) x) (\f2. \x2. f2 (f2 x2)) (\f3. \x3. f3 (f3 (f3 x3)))
= ( \n. \f. \x. (\f2. \x2. f2 (f2 x2)) (n f) x) (\f3. \x3. f3 (f3 (f3 x3)))
=  ( \f. \x. (\f2. \x2. f2 (f2 x2)) ((\f3. \x3. f3 (f3 (f3 x3))) f) x)
=   ( \f. \x. (\x2. x(x x2)) ((\f3. \x3. f3 (f3 (f3 x3))) f))
=   ( \f. \x. ( x(x ((\f3. \x3. f3 (f3 (f3 x3))) )) f))
=   ( \f. \x. ( x(x (( \x3. f (f (f x3))) )) ))
```

4.3 Examples of Lambda Calculus

Here are some examples of Lambda Calculus Reduction in Arithmetic Expression which I calculated by hand.

The first example:

```
(\x. \y. y x) (6 + 2) \x. x + 1 = (\x. \y. y x) 8 \x. x + 1
          = (\y. y 8) \x.x + 1
          = (\x .x + 1) 8
          = 8 + 1
          = 9
```

The second example:

```
(\f. f 5) ((\x. x x) \y. y) = (\f. f 5) ((\y. y) (\y. y))
         = (\f. f 5) (\y. y)
         = (\y. y) 5
         = 5
```

5 Project

For the project, I compare bubble sort in Haskell and C++.

Bubble Sort in Haskell:

```haskell
module Main where
main = do
    contents <- readFile "data.txt"
    print "Data loaded. Doing Bubble Sorting.."
    let newcontents = bubblesort contents
    writeFile "bubblesorted.txt" newcontents
    print "Sorting done"
bubblesort :: (Ord a) => [a] -> [a]
bubblesort []    = []
bubblesort (x0:xs) = case bubble x0 xs of
   (xs', True) -> bubblesort xs'
   (xs', False) -> xs'
   where bubble x1 (x2:xs) | x1 <= x2 = merge x1 False $ bubble x2 xs
                           | otherwise = merge x2 True $ bubble x1 xs
         bubble x1 []               = ([x1], False)
         merge x s (xs, s') = (x:xs, s || s')
```

Compile: ghc -o bubblesort bubblesort.hs
Run: ./bubblesort

Initially, I have tested with these unsorted numbers: 285137496134563242345 from a file called "data.txt". I named the file for this code as "bubblesort.hs". Before "bubblesort.hs" starts sorting, this program has to read a file inside the same folder with the code. Once the bubble sort is complete, the sorted numbers will be place into a file called "bubblesorted.txt". The file "bubblesorted.txt" is created by code of this program. The sorted numbers are 112223333444455566789.

When I test Bubble Sort Algorithm in C++, I also have placed 1000 and 100000 numbers to the txt file. When I placed 1000 numbers, the CPU percentages rise up to 16 percent from 7 percent. After the sorting ends, the CPU percentage goes back to 7 percent. I also notice that the disk percentages fluctuates between 1 and 2 percent when the program is sorting. When I placed 100000 to the txt file, the CPU percentage rises up to 40 percent from 7 percent. After the sort ends, the CPU percentage goes back to 13 percent. I also notice that the disk percentages fluctuates between 1 and 2 percent when the program is sorting. The process for 100000 numbers took over five minutes while, 1000 numbers took less than a minute. I can

conclude that the greater the input, the greater CPU and disk percentage, and the slower the program runs.

Bubble Sort in C++:

```
#include <ctime>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <random>
#include <chrono>
#include <cstdint>
#include <cstring>

using namespace std;
typedef double* DblPtr;

void generateInputFile(const char* filename, size_t size);
DblPtr fillArray(ifstream&, size_t & size);
void bubbleSort(double sortArr[], int size);
void InsertionSort(double arr[], int numbersSize);

int main(int argc, const char* argv[]) {
  int userInput;
  cout << "Type how much data you want to sort: ";
  cin >> userInput;
    generateInputFile("input.txt", userInput);

    if (argc != 2) {
        cout << "Invalid number of command line arguments. Usage:" << endl;
        cout << "Argument 1: input_filename" << endl;
        return -1;
    }

    ifstream inputFile;

    inputFile.open(argv[1]);

    if (inputFile.is_open() == false) {
        cout << "Unable to open input file \"" << argv[1] << '"' << endl;
        return -2;
    }

    time_t startTime, endTime;
    double seconds;
    size_t arrSize;
    DblPtr arr = fillArray(inputFile, arrSize);
    DblPtr copy = new double[arrSize];

    for (size_t size = 1000; size <= arrSize; size *= 10) {

        cout << "-----------------------------------------------" << endl << endl;
        cout << "Array size: " << size << endl << endl;

        memcpy(copy, arr, size * sizeof(double));
        time(&startTime);
        cout << "bubbleSort Start Time: " << startTime << " seconds since Jan 1, 1970" << endl;
```

```
        bubbleSort(copy, (int)size);
        time(&endTime);
        cout << "bubbleSort End Time: " << endTime << " seconds since Jan 1, 1970" << endl;
        seconds = difftime(endTime, startTime);
        cout << seconds << " seconds elapsed." << endl << endl;

    }

    delete[] copy;
    copy = nullptr;
    delete[] arr;
    arr = nullptr;

    return 0;
}

void generateInputFile(const char* filename, size_t size){
    ofstream outputFile(filename);

    outputFile << size << endl;

    unsigned seed = (unsigned)std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator(seed);
    std::uniform_real_distribution<double> distribution(0.0, 100.0);

    outputFile << setprecision(10);
    for (size_t i = 0; i < size; i++) {
        outputFile << distribution(generator) << endl;
    }

}

DblPtr fillArray(ifstream& fin, size_t& size) {
    fin >> size;
    DblPtr arr = new double[size];
    for (size_t i = 0; i < size; ++i) {
        fin >> arr[i];
    }
    return arr;
}

void bubbleSort(double sortArr[], int size) {
    while (size-- > 1){
        double temp = 0.0;
        for (int j = 0; j < size; ++j) {
            if (sortArr[j] > sortArr[j + 1]) {
                temp = sortArr[j + 1];
                sortArr[j + 1] = sortArr[j];
                sortArr[j] = temp;
            }
        }
    }
}


/**
```

```
Compile: g++ bubbleSort.cpp -o bubbleSort
Run: ./bubbleSort input.txt
**/
```

The code for Bubble Sort Algorithm first reads a file "input.txt" that was inputed from the run command. If the file cannot be read, then it returns an error and ends the program. Then it prompts the user to type a number to determine how much to sort. When it starts sorting, it returns the start time. When it finishes sorting, it returns the end time and how much time was elapsed. The start and return time is in seconds since January 1, 1970.

When I test Bubble Sort Algorithm in C++, I have inputed two numbers to be sorted. The numbers that I used are 1000 and 100000. When I inputed 1000 to the Ubuntu terminal, the CPU percentages rise up to 31percent from 19 percent. After the sorting ends, the CPU percentage goes back to 19 percent. I also notice that the disk percentages fluctuates between 5 and 7 percent when the program is sorting. It is noted that my elapse time is 0 seconds elapsed. When I input 100,000 to the Ubuntu terminal, the CPU percentage rises up to 37 percent from 19 percent. After the sort ends, the CPU percentage goes back to 19 percent. I also noticed that the disk percentage goes up to 14 percent and also goes down to 3 percent. It is also noted that when the sorting reaches 1000, the elapse time is 0 seconds elapsed. When the sorting reaches 10000, the elapse time is 1 second elapsed. When the the sorting reaches 100000, the elapse time is 59 seconds elapsed. I can conclude that the greater the input, the greater CPU and disk percentage, and the slower the program runs.

Quick Sort in Haskell:

```haskell
module Main where
main = do
    contents <- readFile "data.txt"
    print "Data loaded. Doing Quick Sorting.."
    let newcontents = quicksort contents
    writeFile "quicksorted.txt" newcontents
    print "Sorting done"
quicksort :: Ord a => [a] -> [a]
quicksort []    = []
quicksort (p:xs) = (quicksort smaller) ++ [p] ++ (quicksort bigger)
    where
        smaller = filter (< p) xs
        bigger = filter (>= p) xs
```

Compile: ghc -o quicksort quicksort.hs
Run: ./quicksort

Initially, I have also tested with these unsorted numbers: 285137496134563242345 from a file called "data.txt". I named the file for this code as "quicksort.hs". Before "quicksort.hs" starts sorting, this program has to read a file inside the same folder with the code. Once the quick sort is complete, the sorted numbers will be place into a file called "quicksorted.txt". The file "quicksorted.txt" is created by code of this program. The sorted numbers are 112223333444455566789.

When I test Quick Sort Algorithm in Haskell, I also have placed 1000 and 100000 numbers to the txt file. When I placed 1000 numbers to the txt file, the CPU percentages rise up to 10 percent from 8 percent. After the sorting ends, the CPU percentage goes back to 8 percent. I also notice that the disk percentages fluctuates between 1 and 2 percent when the program is sorting. When I input 100000 to the txt file, the CPU percentage rises up to 26 percent from 8 percent. After the sort ends, the CPU percentage goes back to 8 percent. I also notice that the disk percentages fluctuates between 1 and 2 percent when the program

is sorting. Both 1000 and 100000 numbers took less than a minute to sort. I can conclude that the greater the input, the greater CPU and disk percentage, and the slower the program runs.

Quick Sort in C++:

```
#include <ctime>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <random>
#include <chrono>
#include <cstdint>
#include <cstring>

using namespace std;
typedef double* DblPtr;

void generateInputFile(const char* filename, size_t size);
DblPtr fillArray(ifstream&, size_t & size);
void InsertionSort(double arr[], int numbersSize);
int Partition(double arr[], int lowIndex, int highIndex);
void Quicksort(double arr[], int lowIndex, int highIndex);

int main(int argc, const char* argv[]) {
  int userInput;
  cout << "Type how much data you want to sort: ";
  cin >> userInput;
    generateInputFile("input.txt", userInput);

    if (argc != 2) {
        cout << "Invalid number of command line arguments. Usage:" << endl;
        cout << "Argument 1: input_filename" << endl;
        return -1;
    }

    ifstream inputFile;

    inputFile.open(argv[1]);

    if (inputFile.is_open() == false) {
        cout << "Unable to open input file \"" << argv[1] << '"' << endl;
        return -2;
    }

    time_t startTime, endTime;
    double seconds;
    size_t arrSize;
    DblPtr arr = fillArray(inputFile, arrSize);
    DblPtr copy = new double[arrSize];

    for (size_t size = 1000; size <= arrSize; size *= 10) {

        cout << "-----------------------------------------" << endl << endl;
        cout << "Array size: " << size << endl << endl;

        memcpy(copy, arr, size * sizeof(double));
```

```
        time(&startTime);
        cout << "Quicksort Start Time: " << startTime << " seconds since Jan 1, 1970" << endl;
        Quicksort(copy, 0, (int)size - 1);
        time(&endTime);
        cout << "Quicksort End Time: " << endTime << " seconds since Jan 1, 1970" << endl;
        seconds = difftime(endTime, startTime);
        cout << seconds << " seconds elapsed." << endl << endl;

    }

    delete[] copy;
    copy = nullptr;
    delete[] arr;
    arr = nullptr;

    return 0;
}

void generateInputFile(const char* filename, size_t size){
    ofstream outputFile(filename);

    outputFile << size << endl;

    unsigned seed = (unsigned)std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator(seed);
    std::uniform_real_distribution<double> distribution(0.0, 100.0);

    outputFile << setprecision(10);
    for (size_t i = 0; i < size; i++) {
        outputFile << distribution(generator) << endl;
    }

}

DblPtr fillArray(ifstream& fin, size_t& size) {
    fin >> size;
    DblPtr arr = new double[size];
    for (size_t i = 0; i < size; ++i) {
        fin >> arr[i];
    }
    return arr;
}


int Partition(double arr[], int lowIndex, int highIndex) {
    // Pick middle element as pivot
    int midpoint = lowIndex + (highIndex - lowIndex) / 2;
    double pivot = arr[midpoint];
    double temp = 0;
    bool done = false;
    while (!done) {
        // Increment lowIndex while numbers[lowIndex] < pivot
        while (arr[lowIndex] < pivot) {
            lowIndex += 1;
        }
```

```
        // Decrement highIndex while pivot < numbers[highIndex]
        while (pivot < arr[highIndex]) {
            highIndex -= 1;
        }

        // If zero or one elements remain, then all numbers are
        // partitioned. Return highIndex.
        if (lowIndex >= highIndex) {
            done = true;
        }
        else {
            // Swap numbers[lowIndex] and numbers[highIndex]
            temp = arr[lowIndex];
            arr[lowIndex] = arr[highIndex];
            arr[highIndex] = temp;

            // Update lowIndex and highIndex
            lowIndex += 1;
            highIndex -= 1;
        }
    }

    return highIndex;
}

void Quicksort(double arr[], int lowIndex, int highIndex) {
    // Base case: If the partition size is 1 or zero
    // elements, then the partition is already sorted
    if (lowIndex >= highIndex) {
        return;
    }

    // Partition the data within the array. Value lowEndIndex
    // returned from partitioning is the index of the low
    // partition's last element.
    int lowEndIndex = Partition(arr, lowIndex, highIndex);

    // Recursively sort low partition (lowIndex to lowEndIndex)
    // and high partition (lowEndIndex + 1 to highIndex)
    Quicksort(arr, lowIndex, lowEndIndex);
    Quicksort(arr, lowEndIndex + 1, highIndex);
}

/**
Compile: g++ quicksort.cpp -o quicksort
Run: ./quicksort input.txt
**/
```

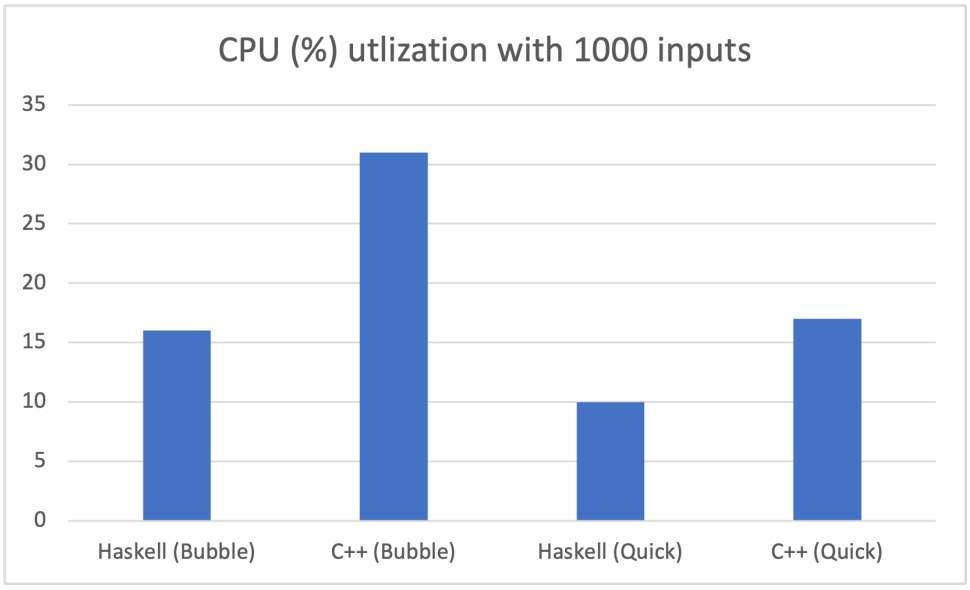
The code for Quick Sort Algorithm first reads a file "input.txt" that was inputed from the run command. If the file cannot be read, then it returns an error and ends the program. Then it prompts the user to type a number to determine how much to sort. When it starts sorting, it returns the start time. When it finishes sorting, it returns the end time and how much time was elapsed. The start and return time is in seconds since January 1, 1970.

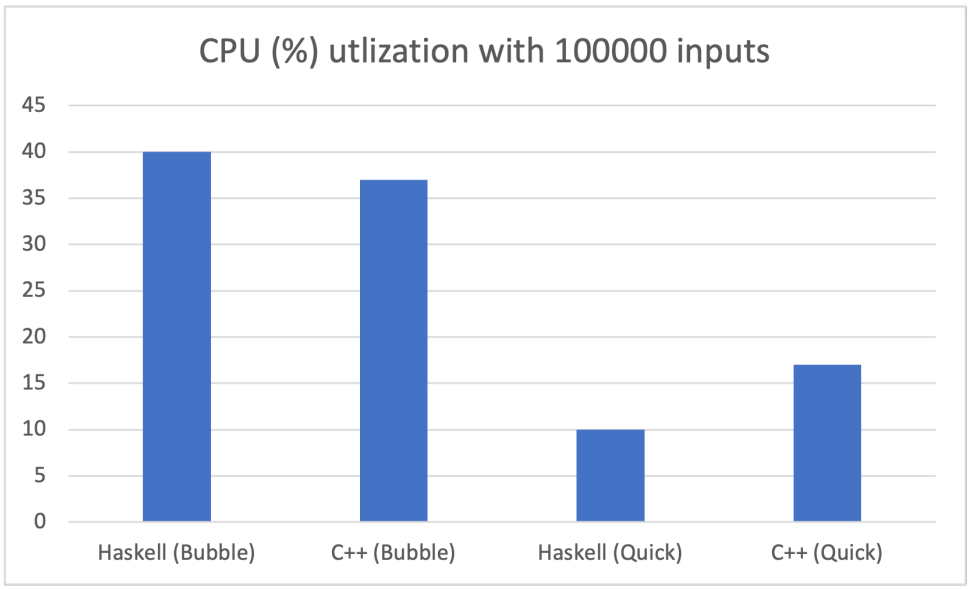
When I test Quick Sort Algorithm in C++, I also have inputed two numbers to be sorted. The num-

bers that I used are 1000 and 100000 which is the same as the Bubble Sort Algorithm in C++. When I inputed 1000 to the Ubuntu terminal, the CPU percentages rise up to 17 percent from 12 percent. After the sorting ends, the CPU percentage goes back to 12 percent. I also notice that the disk percentages fluctuates between 3 and 5 percent when the program is sorting. It is noted that my elapse time is 0 seconds elapsed. When I input 100000 to the Ubuntu terminal, the CPU percentage rises up to 31 percent from 13 percent. After the sort ends, the CPU percentage goes back to 13 percent. I also noticed that the disk percentage goes up to 14 percent and also goes down to the 4 percent. It is also noted that when the sorting reaches 1000, the elapse time is 0 seconds elapsed. When the sorting reaches 10000, the elapse time is also 0 second elapsed. When the the sorting reaches 100000, the elapse time is 0 seconds elapsed again. I can conclude that the greater the input, the greater CPU and disk percentage, and the slower the program runs.

5.1 Project Conclusion

I can conclude that Haskell is much faster than C++ in sorting algorithms. However when I placed 100,000 numbers for Bubble Sort, C++ is much faster than Haskell. For both programs Quick Sort is much faster than Bubble Sort. The percentage usage for CPU and disk is the higher in C++ than Haskell when sorting using both algorithms. However, I see that Haskell runs much faster than C++. Although Haskell runs faster than C++, for large inputs C++ can be faster than Haskell.





6 Conclusions

After writing programs in Haskell and C++ and cosmparing their performance, I found that Haskell is much more complicated than C++. The reason Haskell is more complicated is that I spend a lot of time figuring out how Haskell works and debugging the code to work properly.

References

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

[PL] Lazy Haskell Evaluation Concept, Lazy Haskell Evaluation.

[PL] Faster Code with Laziness, Advantages and Disadvantages of Lazy Haskell Evaluation.

[PL] Ambiguity in Grammar, Ambigious Grammar.

[PL] Immutability is Awesome, Immutable.

[PL] Learning Recursion, Recursion.

[PL] Learning Haskell, Learn Haskell.

[PL] Laziness, Lazy Haskell.