CPSC-354 Report

Anthony Walujono
Chapman University

December 10, 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 Differences with Other Languages

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 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
```

- 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. Compare to Phyton codes take several lines.

1.2 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:

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

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

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.

1.3 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.

1.4 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).

1.5 Problems with Haskell

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.

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: is a context free grammar that has more than one derivation tree.

Example of Ambiguous grammar:

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

Parsing this arithmetic expression 10 - 1 * 5

```
     Exp
   /     \
  Exp  -   Exp
 /    \     |
Exp * Exp  Int
|       |
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.
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     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

The Lambda (represented by symbol in this report) Calculus is the smallest and simplest programming language. The 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))) )) ))
```

5 Regard The Following Contents (Will Be Deleted Soon)

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

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

5.1 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

5.2 Plagiarism

To avoid plagiarism, make sure that in addition to [PL] you also cite all the external sources you use.
This section will contain your own introduction to Haskell.

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.

```haskell
-- 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).

6 Programming Languages Theory

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

7 Project

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

8 Conclusions

Short conclusion.

References

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