Michael Camara

Honor Code Pledge: This work is mine unless otherwise cited

CMPSC 220

Due Date: 10/22/15

Lab 6: Haskell

1) **Typing**

Even though Haskell does not always require explicit type declarations, it is still considered a strongly typed language. Haskell uses a sophisticated system of type inference to first determine the types of values coded by a programmer. It then enforces certain rules that prevent operations from being performed on types not intended by the language. For instance, it will throw an error if a boolean is divided by an integer, or if an exponentiation operator is used on a list. Haskell is further considered statically typed, meaning that the strong type checking is performed at compile time instead of at run time. This helps reduce the incidence of bugs found in Haskell programs, since most problems with type compatibility are identified before such programs are even executed.

2) **Referential transparency**

In general, referential transparency refers to the property of an expression yielding the same output for every input, regardless of its location in a program or time of execution. Further, the official Haskell wiki mentions that this allows for equational reasoning on the code, whereby an expression can be replaced with its value seamlessly. The wiki provides the following example: "if y = f x and g = h y y then we should be able to replace the definition of g with g = h (f x) (f x) and get the same result; only the efficiency might change." This

transparency then allows each function to be evaluated for correctness more easily. It can further help programmers understand and rationalize the behavior of any function without having to consider unclear side effect.

Source: Haskell Wiki. (2014). "Functional Programming." Retrieved Oct. 20, 2015 from https://wiki.haskell.org/Functional_programming#Referential_transparency.

3) **Exponentiation operators**

There are four defined exponentiation operators in Haskell: *exp*, (^), (^^), and (**). The *exp* function uses one argument while the rest use two. The Haskell 2010 Language Report defines these four operators as follows: "The one-argument exponential function exp and the logarithm function log act on floating-point numbers and use base e. […] (^) raises any number to a nonnegative integer power, (^^) raises a fractional number to any integer power, and (⋆⋆) takes two floating-point arguments. The value of x^0 or x^^0 is 1 for any x, including zero; 0⋆⋆y is 1 if y is 1, and 0 otherwise." This may help enforce correct typing for each operator, ensuring that the result of the expression is always of a definitive type. For instance, the (^) operator will always result in integer values while the others will result in floating point values. Here are some further examples showing the output format and certain restrictions for some of the operators:

Allowed	Result	Not Allowed	Reason
5 ^ 2	25	5 ^ 2.5	Exponent is fractional
5.5 ^ 2	30.25	5 ^ (-2)	Exponent is negative
5 ^^ 2	25.0	5 ^^ 2.5	Exponent is fractional
5.5 ^^ 2	30.25		
5 ^^ (-2)	4.0e-2		
5 ** 2	25.0		
5.5 ** 2	30.25		
5 ** (-2)	4.000000000000001e-2		
5 ** 2.5	55.90169943749475		
exp 2	7.389		
exp 2.5	12.182493960703473		
exp (-2.5)	8.20849986238988e-2		

Source: Marlow, S. (2010). *Haskell 2010 Language Report*. Retrieved from https://www.haskell.org/onlinereport/haskell2010/haskellch6.html

4) **Operator Precedence**

The Haskell 2010 Language Report further clarifies the language's operator precedence using the following table:

Prec-edence	Left associative operators	Non-associative operators	Right associative operators
9	`!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``		
6	`+`, `-`		
5			`:`, `++`
4		`==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` ``, `` `notElem` ``	
3			`&&`
2			`||`
1	`>>`, `>>=`		
0			`$`, `$!`, `` `seq` ``

Source: Marlow, S. (2010). *Haskell 2010 Language Report*. Retrieved from https://www.haskell.org/onlinereport/haskell2010/haskellch4.html.

5) **Lists**

Lists are built-in data structures for both Haskell and Python. Both languages support a variety of list operations like concatenation, in addition to allowing complex list comprehension. However, lists in Haskell must be homogenous, whereas lists in Python can be heterogeneous. In other words, every element in a Haskell list must be of the same types (e.g. all integers or all strings), whereas a Python list can have any number of different types contained in it. This choice likely follows the strong typing implementation of Haskell to help prevent against possible type incompatibility while coding.

6) **Reflection**

Most interesting: The ability to have list comprehension that can significantly reduce the amount of code needed to accomplish the same thing in other languages, like Java. For instance, generating an infinite list of even numbers can be done in one line of code: evens = [n | n <- [2..], n `mod` 2 == 0]). Wow!

Hardest to understand: I found it a bit difficult to understand tuples and how I would effectively use them while programming. However, it might just take more experience until I'm comfortable with them.

Most frustrating: So many things! Since I've almost entirely programmed in object-oriented languages up until this point, I expect certain features that a functional language like Haskell simply doesn't provide. For instance, I keep thinking that I can store variables easily, or even

use a simple counter for iteration purposes. I further have to change my normal pattern of using loops to using recursion, which has been quite the challenge.

Most Fun: Reading the “Learn You a Haskell For Great Good” tutorial has been the most fun I’ve had while learning Haskell. I cracked up while reading about the restriction on variables: “If you say that a is 5, you can't say it's something else later because you just said it was 5. What are you, some kind of liar?” I can’t say that I’ve had too much fun while actually programming in Haskell, but I’ve certainly learned a lot during the process.