# CPSC-354 Report

Samuel Ellenhorn
Chapman University

December 27, 2021

**Abstract**

There are many unique theoretical aspects of Haskell which allow for efficient and powerful code. Many of the theoretical choices made in the implementation of Haskell directly allow for it to be capable of what is otherwise impossible in other programming languages. Some of these theoretical ideas are illustrated well in the calculator project. A calculator in some forms can be describes as a basic unit of computation. In Haskell you can build off custom data types to create a completely a unique programming language relatively easily due to its functional nature.The calculator highlights the fact that the choices made in the implementation of any language will determine the languages performance and what it is fundamentally capable of. Haskell's a purely functional nature, and its unique theoretical choices are what set it apart from other established languages. ...

## Contents

# 1 Introduction

Samuel Ellenhorn is a senior majoring in Computer Science. He believes that computers offer extended physical capability which result in potential solutions to problems that would otherwise be insurmountable. When not studying, or building computers, Samuel enjoys volunteering at a local food bank, surfing, playing basketball and listening to music.

## 1.1 General Remarks

In this tutorial we will be using GHC to compile Haskell. GHC has an interactive mode which can be used by typing in ghci in your terminal/command prompt. You can load specific functions from your file using the :I tag before the function name :l myfunctions.

# 2 Haskell

## 2.1 Getting Started:

Open a file in a given directory with the .hs tag at the end (myFile.hs). In the file paste the following function:

```
divTwo x = x / 2
```

1. you can test this out by locating the file with terminal. Next run start the GHC compiler in interactive mode: ghci
2. load the file with: :l myFile.hs
3. call the function: divTwo 8

What does this return? Test the following function:

$$\text{Inc n} = \text{n} + 1$$

What does this return? What are the implications of being able to accomplish this in such a method? Note :

The simple nature of haskell functions play an integral role in its popularity and allow for easy abstraction of complex details. In GHCI, note that you can also calculate divTwo ( Inc 3). Try this. This is a demonstration of encapsulation. The factorial function can be written as follows:

```
Factorial in Haskell:
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

One theoretical aspect of Haskell is the fact that it is a lazy programing language. Because of this, haskell can optimize code to extract redundant calculations. Run the following programs, and compare the amount of time each one took to accomplish the same task.

Fibonacci in Haskell:

```
fib :: Int -> Int

Fib 0 =0
Fib 1 =1
Fib n = fib (n-1) +fib(n+2)
```

In the second version of Fibonacci, we employ the use of haskells built in ability to memoize our algorithm. this can be defined as an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.

Fibonacci in Haskell version 2:

```
import Data.MemoTrie (memo)

fib :: Int -> Int
fib = memo fib'
  where
    fib' :: Int -> Int
    fib' 0 = 0
    fib' 1 = 1
    fib' n = fib' (n - 1) + fib' (n - 2)

main :: IO ()
main = do
  putStrLn (show (fib 30))
```

main :: IO () main = do putStrLn (show (fib 30))

By using a built-in Haskell library MemoTrie, haskell can automatically figures out how to memoize this function. This is as a result of the lazy nature of haskell and is achieved by its robust type system.

## 2.2 Lists:

As you will see, lists in Haskell are strongly and statically typed. This means they won't allow you to

declare a list of different kind of data type. Any list like [3,2,7,4,a,s] will produce the following in the GHCI interactive terminal.

```
<Prelude> [3,2,7,4,a,s]

<Prelude>:1:10: error: Variable not in scope: a

<Prelude>:1:12: error: Variable not in scope: s
```

### 2.2.1 List Concatenation

In Haskell, lists functions work as expected. Lists can be concatenated predictably:

```
<Prelude> [1,2,3,4] ++ [5,6,7,8]
<Prelude>[1,2,3,4,5,6,7,8]
```

The same operations can be applied to strings:

```
<Prelude> "hello" ++ " " ++ "world"
<Prelude> "hello world"
```

This can be accomplished because in Haskell, strings are just lists of characters. If you want to append an element to a list, you can use the : operator as demonstrated here:

```
<Prelude> 6:[1,2,3,4,5]
<Prelude>[6,1,2,3,4,5]
<Prelude> [6,1,2,3,4,5]:7
<Prelude> [6,1,2,3,4,5,7]
```

### 2.2.2 List comprehension:

Lists can be automatically generated using the following short cut producing the following result in the GHCI interactive terminal:

```
<Prelude>[x*2| x<-[1..10]]
<Prelude>[2,4,6,8,10,12,14,16,18,20]
```

### 2.2.3 Other List Methods:

| List Methods | | |
|---|---|---|
| head | :⇔ | takes a list and returns its head. The head of a list is basically its first element. |
| tail | :⇔ | takes a list and returns its tail. In other words, it chops off a list's head. |
| last | :⇔ | takes a list and returns its last element. |
| reverse | :⇔ | reverses a list. |
| sum and product | :⇔ | both take in lists and return the respective sum or product. |
| min and max | :⇔ | takes a list of stuff that can be put in some kind of order and returns the biggest element. minimum returns the smallest. |
| elem | :⇔ | takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way |
| take | :⇔ | takes number and a list. It extracts that many elements from the beginning of the list. Watch. |
| drop | :⇔ | takes number and a list. drops the number of elements from the beginning of a list. |

### 2.2.4 Infinite Lists

Haskell uses a lazy evaluation system, this allows for infinite data structures, however, the compiler will only allocate the terms you use in an expression. In the interactive terminal type [1..] to generate an infinite list of all natural numbers beginning with 1. Try to understand the results of some of the previous list operations on this list. See that by use of the take command, one can dynamical generate a list up to any index:

```
<Prelude>take 10 [1..]
<Prelude>[1,2,3,4,5,6,7,8,9,10]
```

Using infinite lists allows for simple code to perform tasks in such a manner that is not possible in other languages like C++, Java and so on. For example, the factors function takes and integer and in one line of code returns all numbers whos modulus is 0. In the next line we can compute if any number is prime, and return a boolean indicating the answer.

```
Prelude> factors n = [x | x <- [1..n], mod n x ==0]
Prelude> prime n = factors n == [1,n]
```

Below are other methods used to create infinite lists.

| Infinite List Methods | | |
|---|---|---|
| replicate | :⇔ | takes two values n and x and returns a list of length n with x as the value of every element. |
| cycle | :⇔ | takes a list and cycles it into an infinite list. |
| repeat | :⇔ | takes an element and produces an infinite list of just that element. |
| iterate | :⇔ | takes two arguments f and x and returns an infinite list of repeated applications of f to x |

One savvy use of Haskell's unique list comprehension is for the implementation of a binary search tree.

This is gone over in detail in the tutorial at learnyouahaskell.com The code for the BST is implemented here: Haskell-BST for your convenience.

## 2.3 Printing in Haskell

The function **putStrLn** takes a String and displays it to the screen followed by a newline character.The **print** function simply converts an object to a String by way of the show function:

$$\text{print x} = \text{putStrLn (show x)}$$

Because of this, the print function can be used to display non-strings in Haskell. By adding the **deriving (Show)** at the end of a data declaration, Haskell automatically makes that type part of the Show type class

## 2.4 Custom Types and Typeclasses

Every expression and function in Haskell has a type. Type declaration in Haskell allows for the proper allocation of memory. There are three different categories of types: Strongly typed, static types, and types can be automatically inferred. For those with experience in other coding languages, Strongly and statically type declaration should be somewhat familiar[typesH]. However, with type inference; GHCI can automatically guess the types of almost all expressions in a program.Haskell's type system allows us to think at a very abstract level: it permits us to write concise, powerful programs. Types in Haskell are identified with the keyword **data**, this means that we're defining a new data type. The part before the = denotes the type. The parts after the = are value constructors which specify the different values that this type can have. Bellow is some basic examples of creating a Data type, as well as a getter functions and a toString method:

```
data Car = Car {company :: String, model :: String, year :: Int, topSpeed :: Int } deriving (Show)

getCompany :: Car -> String
getCompany (Car getCompany _ _ _) = getCompany

getModel :: Car -> String
getModel (Car _ getModel _ _) = getModel

getYear :: Car -> Int
getYear (Car _ _ getYear _) = getYear

getTopSpeed :: Car -> Int
getTopSpeed (Car _ _ _ getTopSpeed) = getTopSpeed

toStringCar :: Car -> String
toStringCar (Car {company = c, model = m, year = y, topSpeed = t}) = "This " ++ c ++ " " ++ m ++ "
    was made in " ++ show y ++"its top speed is:"++ show t
```

Classes define a set of functions that can have different implementations depending on the type of data they are given. This is a typeclass named BasicEq, instance types within this type class will b referred to with the letter a. An instance type of this typeclass is any type that implements the functions defined in the typeclass. This typeclass defines one function. That function takes two parameters both corresponding to instance types and returns a Boolean.

```
class BasicEq a where
    isEqual :: a -> a -> Bool
    sNotEqual2 :: a -> a -> Bool
instance BasicEq Bool where
    isEqual True  True  = True
```

```
    isEqual False False = True
    isEqual _     _     = False
{-
Testing the above:
isEqual False False should be true
isEqual False True should be False
isEqual True Fasle should be False
isEqual True True should be true
-}
```

### 2.4.1 Monoids

A monoid is when you have a single most natural operation for combining values and a value which acts as an identity with respect to that function. The identity value which does not do anything when you combine it with others. Monoids can be characterized by the following: Firstly, The function takes two parameters(binary function). Second, there exists a value that doesn't change other values when used with the binary function. It is important to note that only concrete types can be made instances of Monoids.[typesH] Lists are an example instance of the Monoid type class. It does not matter what the type of the elements in the list are. Lastly, the parameters and the returned value must have the same type. Monoids must also satisfy the Associative property on the arguments they are called. This means that when there is three or more values and we want to use the binary function to reduce them to a single result, the order in which we apply the binary function to the values doesn't matter. It doesn't matter if we do (2 * 5) * 8 or 2 * (5 * 8). Either way, the result is 80. The same goes for ++ in regards to lists, for example:

$$[0.5, 2.6] + +[0.5, 2.5] + +[0.7, 2.3] = [0.5, 2.5] + +[0.7, 2.3] + +[0.5, 2.6]$$

### 2.4.2 Automatic Derivation

The Haskell compiler can automatically derive instances of Read, Show, Bounded, Enum, Eq, and Ord for many simple data types. Because of this, it is not necessary to manually write code to compare or display custom types. An example of automatic derivation can be seen below:

```
{-Automatic Derivation-}

data Shape = Circle | Box | Triangle
     deriving (Read, Show, Eq, Ord)

{-
Testing
show Circle
(read "Circle")::Shape
(read "[Circle, Box,Triangle]")::[Color]
Box == Box
Box == Triangle
Data.List.sort [Triangle,Box,Triangle,Circle]
 Triangle < Box
-}
```

## 2.5 Monads

Monads are used to contain potentially invalid values in a variable. Monads are built on applicative functors which are built on normal functors. This gives in site to there functionality. In Haskell monads

conceptually are implemented as the **maybe** monad. The **maybe** monad is essentially a form of error checking. In a sense, **maybe** a means that a might be a character, however, it could also be an absence of a character. Monads can be used to represent computations that might have failed. Every monad is an applicative functor. In order to understand Monads, it may be useful to examine what the **maybe** monad type class looks like:

```
class Monad m where
    return :: a -> m a  {-takes a value and wraps it in a Just-}

    (>>=) :: m a -> (a -> m b) -> m b  {-The bind function(>>=) takes a monadic value (that is, a
    value
                                       with a context) and feeds it to a function that takes a
    normal
                                       value but returns a monadic value-}
    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    fail :: String -> m a  {-Directions for Haskell upon failure.-}
    fail msg = error msg
```

Upon analysis, the monad type constructor defines a type of computation. In the above code, the return function creates primitive values of that computation type while the >>= operator is used to combine computations of a that specific type together. This is done too allow for encapsulation in order to make more complex computations.

Monads allow for better code. This is because statements that rely on previous data, do not have to deal with nearly as much error checking. Computations can be composed from simpler sub-computations(encapsulation) which need only take into account proper type casting. Keeping track of types between encapsulated code is the only additional safe guide needed from the actual computations being performed. From a functional programming perspective, this is very good as it requires one only take into account the input and out of sub-components. As a result, programs can be significantly more adaptable than equivalent programs written without monads. This is because the monad factors out many forms of error checking that would otherwise be required to throughout the entire program. These select characteristics of the maybe Monads allow for imperative-style computational structures because functions in your code can remain safely isolated from the main body of the functional program. This is useful for incorporating side-effects (such as I/O) and state (which violates referential transparency) into a pure functional language like Haskell. [monad]

## 2.6 Hanoi in Haskell

Below is a program designed to implement A simple game in Haskell. The game is called Tower of Hanoi, and the premise is to move disks from one peg to a target peg. The target peg is the farthest away and therefore most difficult to get to. While playing you must keep the following in mind.

Rules:

1. Only one disk can be moved among the towers at any given time.
2. No large disk can sit over a small disk.
3. Only the "top" disk can be removed.

Before reading the code it may be helpful to understand the original game itself. You can play the game online here: Play Tower of Hanoi Online

```
hanoi ::
  Int ->
  String ->
  String ->
```

```
  String ->
  [(String, String)]
hanoi 0 _ _ _ = mempty
hanoi n start destination spare =
  hanoi (n - 1) start spare destination   --recursive call
    <> [(start, destination)]
    <> hanoi (n - 1) spare destination start

-------------------------- TEST -------------------------
main :: IO ()
main = putStrLn $ showHanoi 5

------------------------ DISPLAY ------------------------
showHanoi :: Int -> String
showHanoi n =
  unlines $
    fmap
      ( \(from, to) ->
          concat [justifyRight 5 ' ' from, " -> ", to]
      )
      (hanoi n "left" "right" "mid")

justifyRight :: Int -> Char -> String -> String
justifyRight n c = (drop . length) <*> (replicate n c <>)
```

A recursive solution:

1. move the top (N-1) disks to the spare tower
2. move the large bottom disk to the target tower
3. move those (N-1) disk from the spare tower to the target tower.

This recursive solution to the game demonstrates several characteristics of Haskell. Hanoi is a function that takes in 1 int, and 3 strings representing the towers. Hanoi returns a List of pairs containing strings. When printed, these pairs will indicate the moves needed to be taken.

The code treats the tower of disks similar to the stack data structure. The variable n refers to the number of disks in the stack. In the first line, pattern matching is used to check if there are no available moves; in essence it means, if I am moving zero things I have zero moves. If the arguments passed in hanoi match the second hanoi call, a recurssive call is made on a stack of n-1. This means that first recursive call results in the following moves:

$$left -> right$$
$$left -> mid$$
$$right -> mid$$

[hanoi]

## 2.7 Performance Comparison

In this section, we will examine the performance of Haskell in relation to other programming languages. The task of the following code segments is to calculate if a given number is prime or not. The languages used are python, Haskell, and C++.

Python code:

```
import time

# starting time
start = time.time()
num = 100000000**10000000

# To take input from the user
#num = int(input("Enter a number: "))

# define a flag variable
flag = False

# prime numbers are greater than 1
if num > 1:
    # check for factors
    for i in range(2, num):
        if (num % i) == 0:
            # if factor is found, set flag to True
            flag = True
            # break out of loop
            break

# check if flag is True

end = time.time()

if flag:
    print( " true ")
else:
    print( " false ")
# total time taken
print(f"Runtime of the program is {end - start}")
```

Haskell code:

```
factors n = [x | x <- [1..n], mod n x ==0]
prime n = factors n == [1,n]
a=  100000000^10000000

main = do
    print $  prime a
```

C++ code:

```
#include <iostream>
#include <chrono>
#include <stdio.h>
#include <math.h>
using namespace std;
using namespace std::chrono;

// Use auto keyword to avoid typing long
// type definitions to get the timepoint
// at this instant use function now()
auto start = high_resolution_clock::now();
```

```
int main() {
    int i, n;
    bool isPrime = true;

    n=pow(10.0,110.0);

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1) {
        isPrime = false;
    }
    else {
        for (i = 2; i <= n / 2; ++i) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }
    }
    auto stop = high_resolution_clock::now();

    if (isPrime)
        cout << n << " is a prime number";
    else
        cout << n << " is not a prime number";

    auto duration = duration_cast<microseconds>(stop - start);
     cout << "here" << endl;
    cout << duration.count() << endl;
    return 0;
}
```

| Results | | |
|---|---|---|
| python | :⇔ | The highest number capable in a reasonable amount of time is 10**1000000, for this the program took 0.21705198287963867 seconds. |
| Haskell | :⇔ | For the same value $1 * 10^6$, the simple Haskell program took 0.05 seconds, upon further testing 10**100000000 is solvable in under 1.5 seconds |
| C++ | :⇔ | The C++ program had immense trouble with any number larger than 10**100, and it accomplishes this in 2.443914 seconds. |

### 2.7.1 Results Analysis

Do to the fact that Haskell has more optimized native-code compilers it performs faster than Python in many given instances. Python is an interpreted language while Haskell is a compiled language. Both the languages are high-level languages. In regards to the performance comparison between the program in C++ and the program in Haskell, the results are surprising. C is a low level language and is faster than most other languages. In this case, it is likely that the c++ code is too simple and makes use of high level

utilities which negate the advantage of low level language memory control. The program would need to be considerably more complex in order to run faster than Haskell. Due to the many design quirk's as a result of the theory behind Haskell, Programmers are able to create simple and powerful code that is simply not possible in other languages.

# 3 Haskell Theory

## 3.1 Introduction

Haskell is very interesting and unique programming language. There are many reasons why Haskell has taken off as a programming language. First, Haskell is a purely functional programming language that uses pattern matching and lazy evaluation. What makes Haskell so good is that the sub components seem to work well with each other. Lazy evaluations work well with pattern matching, and both allow for a pure functional programming language. In other words functions in Haskell have no internal state. Writing large software systems that work is difficult and expensive. Maintaining those systems is even more difficult and expensive. Functional programming languages, such as Haskell, can make it easier and cheaper. [abtHaskel]

## 3.2 Theoretical Points of Emphasis in Haskell

While there are many reason Haskell as a programming language has seen success, there are several key features regarding the theory behind the programming language that allow for more things to be done easily that would have otherwise been impossible. The following are some theoretical components of Haskell:

### 3.2.1 Purely Functional Language

Functional programming (also called FP) is a method of thinking about software engineering with focus on creating pure functions. It avoids concepts of shared state, as functions do not modify inputs. Because of this, one can understand the entire behavior of a function simply by looking at what the inputs and outputs are. In this sense, A programs is more comparable to a sequence of simple functions rather than a sequence of statements. In most other programming languages, the programmer is to instruct the compiler of a series of tasks and how to carry out these tasks. In Functional Programming languages, there is an emphasis on expressions and declarations rather than execution of statements. Because of this, Haskell does not depend on a local or global state, only values being passed in.

### 3.2.2 Lazy Evaluation

Haskell is a lazy language. This means that Haskell will not evaluate any expression without reason. Another name for this is "Call by name". Call by name is an method of evaluation where the arguments to a function are not evaluated before the function is called. The evaluation is substituted directly into a function body and left to be evaluated whenever it appears in the function. When Haskell's evaluation engine finds that an expression needs to be evaluated, then it creates a thunk data structure to collect all the required information for that specific evaluation and a pointer to that low-level data structure. Haskell's evaluation engine is deferred until the results are needed by other computations. Notably, expressions are not evaluated when they are bound to variables. This is what allows for infinite list comprehensions seen earlier. [HaskellLazy]

### 3.2.3 Haskell as a non-strict Language

One notable characteristic of Haskell is the fact that it can be described as having non-strict semantics by default. A programming language can be defined as having non-strict semantics if expressions can have a

value even if some of their sub expressions have not been evaluated yet. This means that mathematical evaluation proceeds from the outside in. For example given the equation:

(a+(b*c))

The first step is to reduce the +, then you reduce the inner (b*c). This is somewhat counter intuitive but it is factored into the definition of the respective asthmatic. Strict languages work the other way around, they start with the innermost brackets and work outwards.[Non-Strict]

(a+(b*c))

In Haskell, both of these theoretical concepts allow for more flexibly in code. This can be seen in regards to lists. Lists are example of expressions that can contain sub expressions with no value. We know this must be true because as seen before, Haskell allows support for infinite lists. Infinite lists can only be possible if not every value is allocated at run time. The difference between non-strict and Lazy characteristics can be seen in the code below.

```
plusTwo n = n+2.0;
b = -2;
z=3/plusTwo(-2);
a = [5.0,4.0,3.0,plusTwo 1,z,4.0]
elem 5.0 a
```

The above code can be used to illustrate both Lazy evaluation and what it means to be a non strict language. First, it is important to note that the above code should not compile as Z is an invalid value. If the above code is compiled, the call elem 5.0 successfully produces a value of true. You can also call elem on Z, and receive a value of true. Also, you can call elem on 3, and receive a value of true. However when you print the list, the z variable and the expression plusTwo 1 are both evaluated. The fact that the list is evaluated only when printed is an example of lazy programming nature of Haskell. The fact that a list can contain such unevaulated terms such as z and plusTwo 1 are examples of Haskell as a non strict language.

### 3.2.4 Pattern Matching

Haskell supports Pattern matching. In Haskell, pattern matching can be used on any data type. Pattern matching in Haskell consists of specifying patterns to which some data should conform. Haskell will then check to see if the data is conforming, and will then deconstruct the data according to those patterns. An example of a function that uses pattern matching can be seen bellow:

```
module Five where



pull' :: (Ord t, Num t) => t -> [a] -> [a]
pull' k [] = []
pull' k(x:xs)
    | k <= 0 = []
    | otherwise = x: pull '' (k-1) xs
```

If ran correctly, the above function will take in a list and a number, and return a sub-list containing all numbers up to that index. In the function, the first line is an import statement. Line two can be described as the function declaration. Lines three and four are examples of pattern matching. If the function is called on a number, and an empty list, the pattern is matched to line 3, and an empty list is returned. Otherwise, the arguments passed in pattern match to line 4. In this case a recursive call is made. The algorithm is defined to add the head of the list to the subsequent values until the index reaches 0. It is important to

note, that when defining functions, you can define separate function bodies for different patterns as pattern matching is supported for all data types.

Pattern matching leads to a very expressive code. Pattern matching also allows for many convoluted if statements to be inferred by the compiler instead of clouding up code. As a result of this, the code that is produced in Haskell is often times more simple and readable when compared with other languages.

#### 3.2.5 Statically Typed and Type Inference

Haskell is a statically typed language. This means that in Haskell types are known at compile time. This is in contrast to dynamically-typed languages which perform type checking at runtime. In Haskell, you must declare the data types of your variables before you use them. This however comes with a caveat: Haskell can also be described as having full type inference. This means that while in other programming languages, it is necessary to define a series of variables along with their respective type, the same is not always true in Haskell. Haskell also has the ability to employ type inference. Type Inference refers to the ability of the Haskell compiler to automatically detect the type of some expressions. This is made possible in part because Haskell can be described as a strongly typed language. A strongly typed language is one where there are restrictions between conversions between types. These factors contribute to the Haskell compilers capability of figuring out the type of some variables without the code explicitly mentioning them. The following code example found online can provide incite into these theoretical ideas:

```
func baseAmt str = replicate rptAmt newStr
  where
    rptAmt = if baseAmt > 5
      then baseAmt
      else baseAmt + 2
    newStr =  Hello     ++ str
```

In the above Haskell code, types are not assigned to the expressions: “baseAmt” and “str”.This is normal in languages such as Java or C++. This is because the Haskell compiler has enough information to know that “baseAmt” is an integer, and “str” is a string without us telling it. Also, if the result of this function is needed somewhere else in the program, the compiler will already know that the resulting type is a list of strings. This is because of Haskell’s strongly typed nature. Haskell’s statically typed nature and its ability to utilize type inference are foundations of what allow Haskell to make use of the principles of functional programming, as such, code is more likely to consists of expressions to be evaluated.[haskType]

## 3.3 Decidability and Undecidability

Haskell is unique language in the sense that its theoretical components play a huge role in the fundamental ways programmers choose to structure their code. Because of this, it can be use full to examine problems from different perspectives. One such perspective is a theory of computation regarding weather a problem is decidable or not. A problem is considered to be decidable if there exists a Turing machine that will always halt and give the correct answer. In other words a problem is decidable if there exists an algorithm that can be written to solve it every time. [decidable-3] [decidable-2] [decidable-1]

### 3.3.1 Is Haskell A Decidable language?

Given the previous definition, it is clear to see that decidability is a definitive goal of any programming language. With that in mind, it is a bit difficult to determine the truth for such a blanket statement as: ”Is Haskell decidable?”. This statement is similar to asking: ”Can Haskell solve every problem?”

From the perspective of a turing machine, a decidable language is composed of strings that will either halt at the accept or the reject state on each input string. With that in mind, it is useful to ask weather Haskell as a language is decidable for some questions or not. In order to answer this, it is necessary to approach from the perspective of the overall the concept of language. Most programming languages have an ambiguous formal grammar, because ambiguous grammars are easier to write than unambiguous grammar. Ambiguous grammars are context-free grammars for which there exists a string that can have more than one parse tree. These are important aspects of Haskell to keep in mind when exploring this topic.

A decision problem P is decidable if the language L of all yes instances to P is decidable. [decidable-4] In other words, a language is decidable for a given problem if it is a recursive language regrading that problem. This means that all solutions to that problem are made up of strings which exist in the language itself. This is important because it gives incite into the problems that in theory could be solved by Haskell. In fact, it is true that every question about regular language is decidable, however, in context free language like Haskell, some questions are not decidable by nature.

Undecidability, as a concept can be better understood through application to problems as it is a property of problems, not one of programming languages. In computability theory and computational complexity theory, an undecidable problem is a decision problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes-or-no answer[decidable-1]. One example of something that is undecidable is to check if some given program ever halts on any input. This problem takes a program and a programs inputs as its own inputs. This problem is known as the = halting problem. The halting problem is proof of undesirability of some first order logical questions. It basically states that it can be proven that there is no algorithm that correctly determines whether arbitrary programs eventually halt when ran on some arbitrary inputs.

[decidable-3] [decidable-2] [decidable-1] [decidable-4] [decidable-5]

## 3.4 Halting Problem: to Halt or not to Halt

The Halting problem is one of the most well-known problems that has been proven to be undecidable.The Halting problem is a decision problem in compatibility theory. It deals with the following question: Will a particular program halt on a given input or not. That is to say : given a computer program and an input, will the program terminate or will it run forever?

### 3.4.1 Halting Problem Explained: proof by contradiction

In order to understand this problem, it is only important to take into account the inputs and outputs of some arbitrary programs. Firstly, we assume that the halting problem is solvable, meaning there exists some program f which takes in two inputs:

1. An arbitrary computer program.
2. The inputs of the above program.

Program f will then output if the arbitrary program halts with given input or not. Now, if the arbitrary program is designed to specifically do the opposite of what f predicts it will do. No f can exist that handles this case. In other words, if the arbitrary program takes into account the output of the above program f, and simply returns the opposite, we enter an infinite loop. In 1936, Alan Turing proved that the halting problem over Turing machines is undecidable using a Turing machine; that is, no Turing machine can decide correctly (terminate and produce the correct answer) for all possible program/input pairs.[halting1][halting][halting4]

### 3.4.2 Halting Problem Conclusions

Using Alan Turing model for computation it is clear that the halting problem is not solvable. With that in mind it is possible to make progress on this problem by means of another method. We will later examine that is possibly to potentially solve this problem using lambda Calculus, however it requires that our definitions for our language are no longer Turing complete.

## 3.5 Lambda Calculus

Lambda Calculus is a useful method for examining how functions, arguments and values interact. functions are represented using a Lambda expression. A Lambda expression is composed of a head and a body. The notation for Lambda Calculus is as follows:

$$\lambda(head)(body)$$

- The $\lambda$ is sometimes represented with a back slash.
The head holds the function parameters, while the body describes what is returned by the function. Variations and extensions on Lambda Calculus form the basis of Haskell. Lambda expressions are applied to variables. An example of a lambda expression is as follows:

$$(\lambda x.x + 1)5$$

The above function would yield the result 6. This is because we will substitute the 5 in for the for x, this makes the return value $5 + 1$ yeilding 6 as the final answer.

### 3.5.1 Lambda Calculus Examples

The following are some examples of useful functionality implemented in Lambda Calculus:

### 3.5.2 Lambda Calculus Examples

Boolean's in Lambda Calculus:

$$True = (\lambda x.\lambda y)x$$
$$False = (\lambda x.\lambda y)y$$

Not function in Lambda Calculus:

$$Not(!) = \lambda b.bFalseTrue$$

apply to a Boolean True:

$$(\lambda b.bFalseTrue)True$$

substitute True into b:

=

$$TrueFalseTrue$$

expand out True:

=

$$(\lambda x.\lambda yx)FalseTrue$$

Lastly, apply the function to the two values False and True in order to obtain the final value of False.

| Other lambda operations | | |
|---|---|---|
| And Gate | :⇔ | $(\lambda xy.(x(yTrueFalse)False))$ |
| OR Gate | :⇔ | $(\lambda xy.(xTrue(yTrueFalse))$ |
| Multiplication | :⇔ | mult$[\lambda n.\lambda m.m(addn)0]$ |
| Infinite Loop | :⇔ | $(\lambda x.xx)(\lambda x.xx)$ |

### 3.5.3 Evaluation Models

Evaluation Models: There are many different models of evaluation. An evaluation strategy is a set of rules for evaluating everything in a programming language. The evaluation strategy chosen will decide what order the arguments of a function are evaluated and when they are substituted into a function. [**?**] There are three primary strategies: Call-by-value, Call-by-name, and Call-by-need. The following will discuss each in detail.

### 3.5.4 Call-by-value

In the call-by-value method of evaluation; the actual parameter expression is evaluated and then the resulting value is bound to the corresponding formal parameter variable of the function. Arguments are evaluated before a function is entered. An example of Call-by-value method used to expand and solve a lambda expression.[eval-method]

$$\begin{aligned}(\lambda x.\lambda y.yx)(3+3)(\lambda x.x+1)\\ =(\ \lambda x.\lambda yx)6(\lambda x.x+1)\\ =(\ \lambda y.y6)(\lambda x.x+1)\\ =(\ \lambda x.x+1)6\\ =6+1\\ =7\end{aligned}$$

### 3.5.5 Call-by-name

In the call-by-name method of evaluation, the arguments to a function are not evaluated before the function is called. An actual parameter expression is bound to the corresponding formal parameter variable before it is evaluated. In other words, arguments are passed unevaluated. In call-by-name evaluation, the arguments to lambda expressions are substituted as is. Evaluation proceeds from left to right substituting the outermost lambda or reducing a value. If a substituted expression is not used it is never evaluated. An example of call-by-name method used to expand and solve a lambda expression is seen below:[eval-method]

$$\begin{aligned}
&(\lambda x.\lambda y.yx)(3+3)(\lambda x.x+1) \\
&= (\lambda y.y(3+3))(\lambda x.x+1) \\
&= (\lambda x.x+1)(3+3) \\
&= (3+3)+1 \\
&= 6+1 \\
&= 7
\end{aligned}$$

### 3.5.6 Call-by-need

In the call-by-need method of evaluation; arguments are passed unevaluated, however an expression is only evaluated once and is stored for subsequent references. Call-by-need is a more efficent version of call-by-name however it require that all arguments are pure functions. This is because the evaluation of subexpression does not follow any pre-defined order, so any impure functions, will be evaluated in an unspecified order. As a result call-by-need can only effectively be implemented in a purely functional setting.[eval-method] Call-by-need is a special type of non-strict evaluation in which unevaluated expressions are represented by suspensions or thunks which are passed into a function unevaluated and only evaluated when needed or forced. When the thunk is forced the representation of the thunk is updated with the computed value and is not recomputed upon further reference. The thunks for unevaluated lambda expressions are allocated when evaluated. The resulting computed value is placed in the same reference so that subsequent computations share the result. If the argument is never needed it is never computed, which results in a trade-off between space and time. [eval-method]

### 3.5.7 Is Lambda Calculus Turing complete?

Something is considered to be be Turing complete if it can be used to simulate any Turing machine. In order to answer this question it can be useful to examine church encoding. With that in mind, it would seem as though all functionality of a particular programming language can be implemented with Lambda Calculus. There are three properties of Lambda Calculus that allow for this:

1. The ability to create functions out of Lambda.
2. The ability to utilize variable references.
3. The ability to apply functions.

Utilizing these three properties, you can create any program. Therefore, Lambda Calculus is Turing complete.

### 3.5.8 Church Encoding

Church encoding is a process of encoding all values as lambda abstractions.[Church Encoding] Below is an example of code in Haskell Translated into Lambda Calculus:

| Racket to Lamda Calculus | | |
|---|---|---|
| (let ([x 20])(+ x 3)) | :⇔ | ((lambda (x) (+ x 3)) 20) |
| (let ([x ( + 21 ( * 2 3))]) ((lambda (y) y) x)) | :⇔ | ((lambda (x) ((lambda (y) y) x)) ( + 21 ( * 2 3))) |
| (let ([x 20] [y 30]) (+ x y)) | :⇔ | ((lambda (x y) (+ x y)) 20 30) |

Racket is a much simpler and much smaller language than Haskell. If you test the above functions, they will yield the same result. Here is another example including the expansions and substitutions:

Converting Process:

Step 1:

(let ([x ( + 21 ( * 2 3))]) ((lambda (y) y) x))

Step 2: Substitute the lambda by applying the previous function

((lambda (x) ((lambda (y) y) x)) ( + 21 ( * 2 3)))

#### 3.5.9 Conclusions on Lambda Calculus

Lambda Calculus can encode any computation, i.e. any program in any language can be represented in some way by Lambda Calculus. Lambda Calculus is the basis for any functional programming language. Lambda Calculus is present in many programming language. Lambda Calculus can represent any code language, and we know languages like Python and Java are already Turing complete. The lambda calculus is older than the Turing's machine model, dating from the period 1928-1929 (Seldin 2006). Lambda calculus was invented to encapsulate the notion of a schematic functions that ChurchC:.hs needed for a foundational logic he devised. It was not invented to capture the general notion of compatible functions. [Church Encoding 3] [Church Encoding 2]

# 4 Project

For my project I chose to extend the calculator. The calculator includes the fraction and Boolean data type. Fractions can be evaluated into floats. Safe division on natural numbers and the fraction data type have been implemented using a maybe Monad. Several helper functions were also added for operations on fractions. The project extensions takes advantage of several important theoretical aspects of Haskell.

## 4.1 Safe Division

In this example, we can see that the maybe monad is used for safe division. The safediv function takes in two NNs and outputs a Maybe NN. A maybe monad has two values: just which in this case will hold a NN and Nothing. If pattern matching indicates that the second argument is 0, the value of the Maybe NN will be evaluated to Nothing. In other words, if two arguments are given and the second is 0 return Nothing. This is accomplished in the second line of the code below by way of pattern matching. In another case, if two arguments are passed in, and the first is 0 return Just 0. Just is comparable to a container that holds a val 0. If you haven't pattern matched to any of these cases, you reach the case statement. Given two NNs, if argument 1 < argument 2 return 0. This is because the type doesn't support floats. Otherwise the algorithm accomplishes recursive safediv on 1 + (argument 1- argument 2). The following is an example:

$$\frac{10}{4} => 1 + \frac{6}{4} => 1 + 1 + \frac{2}{4}$$

note: $\frac{2}{4}$ evaluates to 0 because of the first case statement.

```
safediv :: NN -> NN -> Maybe NN
safediv _ O = Nothing
safediv O _ = Just O
safediv p q
  | p `less` q = Just O
```

```
  | otherwise = Just (addN (S O) a) where
    Just a = ((subtr p q) `safediv` q)
```

The normalizeI function is a helper for other functions in the calculator. Its functionality is not necessarily intuitive. In order to understand the normalizeI function, it is important to note the data type Integer is defined as follows:

```
data II = II NN NN
deriving (Eq,Show) -- for equality and printing
```

This is because the language has to represent negative numbers by the subtraction of the two NNs. For example: II 7 9 could represent -2, however it would be better to represent -2 as II 0 2. This is the function of the normalizeI function.

```
normalizeI :: II -> II
normalizeI (II a b)
    | grtr a b = II (subtr a b) O
    | otherwise = II O (subtr b a)
```

Using A and B to model the first and second inputs respectively, a simple case statement is used. If A is greater than B, you are dealing with a positive number which is to be stored in the first NN value, otherwise, if A is less than B, you can simplify by subtracting A from B and storing it as the second elem along with 0. Both of the following conversion's utilize recursion and pattern matching. Here we can see why the normalize function is highly useful.

```
ii_int :: Integer -> II
ii_int a
    | a>0 = nn_ii (nn_int a)
    |otherwise = (II O (nn_int(-1*a)))


int_ii :: II -> Integer
int_ii (II a O) = int_nn a
int_ii (II O b) = (-1) * (int_nn b)
int_ii x = int_ii (normalizeI x)
```

The following function will convert fractions into floats. This is accomplished by using Haskell division and the fromIntegral function. Also, it uses the previous conversion functions, so it is important to keep in mind types. $int\_ii$ takes a custom integer and returns a Haskell integer. fromIntegral takes an Haskell integer and turns into a floats. Haskell division also returns a float. The development of this function is made easier by the fact that Haskell is a purely functional programming language. I need only worry about inputs and outputs.

```
float_qq :: QQ -> Float
float_qq (QQ a b) = (fromIntegral(int_ii a)/fromIntegral(int_pp b))
```

## 4.2 Fractions

The following are helper functions for future operations on fractions. p2n recursively converts positive numbers into natural numbers. p2n is called on 1, and the successor of any positive number. n2p is the inverse of p2n and operates using the same method, however; n2p pattern matches out the zero-th case.

```
--helper functioned needed for frac
```

```
-- Convert from PN to NN
p2n :: PP -> NN
p2n I = S O
p2n (T n) = S O `addN` p2n n--add the amount of ones\


n2p :: NN -> PP
n2p O = error "O cannot be coarced to PN"
n2p (S O) = I
n2p (S n) = n2p n `addP` I ---if called on the succser of a number add 1
```

Fraction multiplication is simple enough to implement, it is only necessary to keep track of the denominators and numerators and multiply them respectively. It is important to note that the fraction data type is Defined as follows:

```
data QQ = QQ II PP
```

Each QQ consists of an integer which can represent negative numbers, and a positive denominator which can not equal 0. Notice in the mulF function that because Haskell is a lazy programming language, both num and den are not determined until they are called by the simplifyF function.

```
mulF :: Frac -> Frac -> Frac
mulF (a,b) (c,d) = f where --f will be defined in following lines
  num = a `multN` c  --multiply the numerator
  den = b `multP` d  --mult the denom
  f = simplifyF (num,den)
```

Addition is a bit more interesting as the Frac data type is composed of a natural number and a positive number for the numerator. Because of this it is necessary to convert the denominators and numerator properly.

```
-- addFractions:
addF :: Frac -> Frac -> Frac
addF (a,b) (c,d) = f where
  num =  (multN a (p2n d)) `addN` (multN c (p2n b))
  den = b `multP` d
  f = simplifyF (num,den)
```

Division of fractions can also be seen as practice of keeping track of the types. This is because the simplest way to implement division of fractions is to implement multiplication of the reciprocal. In normal division of allocated as its reciprocal and then multiplied blindly utilizing the mulF function. This can result in division by zero potentially.

```
--divFractions:
divF :: Frac -> Frac -> Frac
divF (a,b) (c,d) = f where --f will be defined in following lines
  fracOne = (a,b)
  fracTwo = (p2n(d), n2p(c))--if C is zero we will end up with an invalid ans
  frac = (fracOne `mulF` fracTwo)
  f = simplifyF(frac)
```

To remedy the short comings of the above function, we can use a maybe monad to error gracefully. The first pattern matching check is on the numerator of the second argument; if this is 0, our result will be Nothing, in other cases, take the same steps as the previous algorithm.

```
--safeDivFrac:
safedivF :: Frac -> Frac -> Maybe Frac
safedivF (a,b) (0,d)  = Nothing--f will be defined in following lines
safedivF (a,b) (c,d) = f where
  fracOne = (a,b)
  fracTwo = (p2n(d),n2p(c))--if C is zero we will end up with an invalid ans
  f = Just(simplifyF(fracOne `mulF` fracTwo))
```

Fraction equality can also be implemented successfully utilizing our language as well. Fractions are seen as equal if the product of the numerator from the first term and denominator from the second term are equal to the product of the numerator from the second term and denominator from the first term. This is a simple and intuitive check. Fraction simplification on the other hand, require a bit more nuance. It is accomplished by finding the gcdN of numerator and type converted denominator. This new numerator and denominator are the original divided by the previous gcdN value. This can be seen in the following code segment.

```
-- Fraction equality
equalF :: Frac -> Frac -> Bool
equalF (a,b) (c,d) = a `multN` (p2n d) == c `multN` (p2n b)

-- Simplify Fractions
simplifyF :: Frac -> Frac
simplifyF (a,b) = (p,q) where
  gcd =  a `gcdN` (p2n b)
  p = a `divN` gcd
  q = n2p $ (p2n b) `divN` gcd
```

## 4.3 Implementation of Boolean logic

The following code segment is the implementation of Boolean logic in Haskell with custom data types. At its core a boolean can either be true or false represented as Tr and F. Operations on bools include or, and and not.

```
data Boole = Tr | F
  deriving (Eq,Show) --


--------------------
--bool addition (or)
--------------------
addBoole :: Boole -> Boole -> Boole
addBoole Tr F = Tr
addBoole F F = F
addBoole Tr Tr = Tr

--------------------
--bool mult (and)
--------------------
multBoole :: Boole -> Boole -> Boole

multBoole F F = F
multBoole Tr F = F
multBoole Tr Tr = Tr
```

```
--------------------
--bool (not)
--------------------
notBoole :: Boole -> Boole
notBoole F = Tr
notBoole Tr = F
```

With the addition of Boolean values, it is possible that case statements(if-then-else) can be implemented successfully. In other programming languages, case statements often use expressions which are based on Boolean values. If some Boolean value is true, then something happens; otherwise, something else can occur.

### 4.4 Project Conclusion

This project extension takes advantage of important theoretical aspects of Haskell. Operations like Safe division have been implemented using a maybe Monad. This only possible if Haskell uses lazy evaluations. When the program is compiled, the monad is not yet defined. In addition, several helper functions were also added for operations on fractions. These helper functions make use of pattern matching, and the functional nature of Haskell as a whole.

## 5 Conclusions

In short, the many unique theoretical aspects of Haskell which allow for efficient and powerful code. Many of the theoretical choices made in the implementation of Haskell directly allow for it to be capable of what is otherwise impossible in other programming languages. Some of these theoretical ideas are illustrated well in the calculator project. A calculator in some forms can be describes as a basic unit of computation. In Haskell you can build off custom data types to create a completely a uniqe programming language relatively easily due to its functional nature.The calculator highlights the fact that the choices made in the implementation of any language will determine the languages performance and what it is fundamentally capable of. Haskell's a purely functional nature, and its unique theoretical choices are what set it apart from other established languages.

## References


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

[hanoi] Hanoi

[Non-Strict] Non-Strict

[video] Haskell style-using pattern matching,

[video] Lambda Calculus - Computerphile

[Doc] Lambda Calculus

[article] haskell VS python

[wiki] Wiki.haskell.org

[wiki] What is a monad?

[tutorial] Main Reference Tutorial

[abtHaskel] About Haskell

[haskType] Haskell Type System

[typesH] Haskell Types

[Church Encoding] Church Encoding

[Church Encoding 2] The Logic of Curry and Church

[Church Encoding 3] A Formulation of the Simple Theory of Types

[Church Encoding 4] Church Encoding 4

[halting1] halting1

[halting2] halting2

[halting] halting

[halting4] halting4

[HaskellLazy] HaskellLazy

[eval-method] eval-method

[eval-method2] eval-method2

[decidable-1] decidable-1

[decidable-2] decidable-2

[decidable-3] decidable-3

[decidable-4] decidable-4

[decidable-5] decidable-5

[decidable-6] decidable-6

[monad] monad