# Purpose

The purpose of this project is to investigate how Haskell works and how its features benefit the developers. This investigation will show why Haskell as a high level language is better than standard imperative languages in terms of safety, expressiveness, high-level abstractions, modularity and performance. A lot of the concepts that Haskell uses can be found in other functional programming languages and even in some imperative languages. Over the course of this project I will demonstrate how Haskell works, explain its concepts, and compare that to more commonly used languages and concepts.

# Analysis

When we pick a programming language for a project we want it to have several features that will make the process of building the project faster and easier. In this investigation project I will identify the most important features that a general purpose high-level programming language should have and show how Haskell implements them. First, let's identify the main features that make high-level programming languages different from the low-level ones.

## Goal

### Safety

Usually programmers spend a lot of time finding and fixing bugs, which is a huge problem because for businesses it's crucial to deliver code as fast as possible. But what's an even bigger problem is programs crashing in production. This can cause businesses to lose a lot money or, in the worst cases, shut down. So we want programming languages to prevent as many bugs as possible.

A lot of run-time issues happen because of state changes. In most languages variables are mutable and functions can have side effects, this means that we can accidentally change a global variable and cause the entire program to crash.

Here are some examples of methods that we can use to solve this problem: type systems, automated tests, code checkers. Let's take a look at every one of these approaches. Not all languages have type systems, but even the ones that do usually allow dangerous actions like implicit casting. Automated tests are great and they help a lot, but it's hard to test every single thing, especially if the part of the code that you are testing depends on the state. Non-standard code checkers can help, but often they can't prevent even simple run-time errors.

### Expressiveness

We want to write the least amount of code possible. At the same time we want the code to be readable and elegant. As I mentioned above the development speed is very important and it reduces if we can quickly express ideas without writing too much code.

To solve this problem most languages just create special syntax for common things, for example instead of just having while loops they also support for loops. This can help, but this approach has two problems: it clutters up the language and it doesn't solve the problem generally, this approach only adds hacks for common specific things.

### High level abstractions

High-level languages allow developers to think just about solving the business problem and minimize the amount of the code that specifies what exactly happens inside the computer. There are two areas that are fundamental in programming - resource management and sequencing. Most modern languages have tools like garbage collectors that automatically deal with resource management, however, as most languages are imperative, they force the programmer to specify the order of computations. This is because the main idea of imperative programming is to allow programmers to write a sequence of instructions that computer will execute. The only thing imperative languages can do to abstract sequencing is introducing new keywords and libraries, thus cluttering the language.

We also want to be able to work with complex abstractions and general versions of different types. For example if the only important aspect of a data structure is that it can be parsed from a string then we don't want to work with a specific type, we want to use the most general case possible. Also this helps avoiding errors.

### Modularity

We want to reuse as much code as possible. We also should be able to easily compose different parts of a program together. This is one of the main concepts of programming.

### Performance

Depending on what we are developing we have different performance requirements, so we want programming languages to help us make efficient programs. Of course using a high-level language for writing low-level applications like drivers is usually a bad idea, but high-level languages should work as fast as possible without loosing benefits of high-level languages.

## How Haskell works

### Hello World and more.

main :: IO ()  
main = print "Hello World!"

Now let's write a console program that asks user to input their name and then prints "Hello <username>!" Haskell's IO parts of the program can look very similar to "normal" programming languages.

main :: IO ()  
main = do  
 print "What's your name?"  
 name <- getLine  
 print ("Hello " ++ name ++ "!")

But we can write this in functional style.

main :: IO ()  
main = print "What's your name?" >> getLine >>= print . (++ "!") . ("Hello " ++)

We will come back to both of these examples later.

### Function purity

Haskell is very different from most languages. In Haskell all variables are immutable. This means that you don't really have variables, you only have constants. Also in Haskell all functions are pure. A pure function a function that any time it gets called with the same arguments returns the same result. Pure functions don't have side effects; they can't print something to console, read files or modify variables. Functions in Haskell are like functions in maths, they are just mappings between types. These properties make testing and debugging code much easier.

### Lazy evaluation

Another aspect that makes Haskell very different from an average programming language is the fact that by default it uses lazy evaluation. This means that functions won't get evaluated until the result is needed. When a program gets executed it won't do unnecessary computations.

### Defining functions

Let's define a function f that squares a number in both Python and Haskell. Here is how it would look like in Python:

def f(x, y):  
 return x\*x + y\*y

And here is the Haskell version:

f x y = x\*x + y\*y

In Haskell to pass arguments into a function we don't use brackets and/or commas, we separate arguments with spaces. As you can see the definition is very simple and it doesn't use any unnecessary syntax like def or return. It's just the function name, arguments and what it returns.

In Haskell functions and types are the two primary things and everything is centered around them, so it makes sense why it's very easy to define them.

### Introduction to the type system

In Haskell you don't need to explicitly declare types of functions or variables, the compiler will derive them for you. However, explicitly declaring types of functions and variables is a good practice. Let's declare the type of the previous function and then write a main function to test f.

f :: Int -> Int -> Int  
f x y = x\*x + y\*y  
  
main = print (f 2 3)

But what if we want function f to work with all numbers and not just integers. The first solution is to remove the type declaration, in that case our file would look like this:

f x y = x\*x + y\*y  
  
main = print (f 2.1 4)

GHC (Glasgow Haskell Compiler) is the default Haskell compiler. Haskell can be both compiled and interpreted, which is why there is an interactive environment - GHCi, which you can use to run Haskell code without making a file for it. It can also tell us the type of any defined function. Let's use it to find the type of f.

Prelude> :load sum\_squares.hs   
[1 of 1] Compiling Main ( sum\_squares.hs, interpreted )  
Ok, modules loaded: Main.  
\*\*Main> :t f  
f :: Num a => a -> a -> a  
\*\*Main>

OK, let's figure out what that type is.

|  |  |
| --- | --- |
| Type | Value |
| / | <> |
| Int | An integer |
| Int -> Int | A function that takes an integer and returns an integer |
| Float -> Int | A function that takes a float and returns an integer |
| a -> Int | A function that takes a value of any type and returns an integer |
| a -> a | A function that takes a value of any type and returns something of the same type |

In Haskell type a -> a -> a is the same as a -> (a -> a). This means that this is a function that takes an argument of any type and returns a function that takes an argument of the same type and returns something of the same type, so basically it's a function with two arguments. The benefit of this representation is that we can give the function only one argument and get a valid expression which is a function. This is called partial application.

When in a type declaration you see something starting with a small letter, it means that it's a type variable. Type variables give us parametric polymorphism. Also, for example, if you have a function that takes two arguments of any type, but both arguments have the same type, you can specify that using type variables.

But our function type is not just a -> a -> a, it also has prefix Num a =>. This means that a is in the type class Num. Type classes are like interfaces in OOP languages. They declare a list of signatures of variables, functions, and types. A type is in a type class if it implements all the members of the type class.

class Num a where  
 (+) :: a -> a -> a  
 (-) :: a -> a -> a  
 (\*) :: a -> a -> a  
 negate :: a -> a  
 abs :: a -> a  
 signum :: a -> a  
 fromInteger :: Integer -> a

Here is the definition of the type class Num. In Haskell operators are just normal functions. By writing Num a => we restrict all possible types to only allow the ones that implement the functions listed above.

So the type Num a => a -> a -> a means that it's a function that takes a number and returns a function that takes another number of the same type and then returns a number of the same type. Technically all functions in Haskell take only one argument. But any function that takes two arguments can be represented as a function that takes one argument and returns a function. So the expression f 3 4 is equivalent to (f 3) 4 and f 3 is a function.

To define functions we can use another notation - lambda functions.

f = \x y -> x\*x + y\*y

### Basic minimum of Haskell

I will use <=> to show that two expressions are equivalent. This is not a part of the Haskell syntax.

1. Arithmetic operations

* 3 + 2 \* 6 / 3 <=> 3 + ((2 \* 6) / 3)

1. Logic

* True || False <=> True  
  True && False <=> False  
  True == False <=> False  
  True /= False <=> True

1. Powers

* x ^ n -- for non-negative integer powers  
  x \*\* y -- for floating numbers

1. Lists

* [] -- empty list  
  [1, 2, 3] -- a list of numbers  
  ["foo", "bar"] -- a list of strings  
  1:[2, 3] <=> [1, 2, 3] -- (:) prepends an element to a list  
  1:2:[] <=> [1, 2]  
  [1,2] ++ [3,4] <=> [1, 2, 3, 4] -- (++) joins two lists  
  [1,2] ++ ["?"] -- compilation error  
  [1..4] <=> [1, 2, 3, 4]  
  [1,3..10] <=> [1, 3, 5, 7, 9]  
  [2,3,5,7..100] -- error, the compiler is not that smart  
  [5,4..1] <=> [5, 4, 3, 2, 1]

1. Strings

* In Haskell strings are just lists of chars.
* 'a' :: Char  
  "a" :: [Char] -- :: String  
  "ab" -- ['a', 'b']
* This is not very efficient, which is why in most cases people use other data types that represent strings.

1. Tuples

* -- All of these tuples are valid  
  (2,"foo")  
  (3,'a',[2,3])  
  ((2,"a"),"c",3)  
    
  fst (x, y) = x  
  snd (x, y) = y  
    
  fst (x, y, z) -- ERROR: fst :: (a, b) -> a  
  snd (x, y, z) -- ERROR: snd :: (a, b) -> b

### Applying functions

Here are two operators that are used very often.

(.) :: (b -> c) -> (a -> b) -> a -> c  
(.) f g x = f (g x)  
  
($) :: (a -> b) -> a -> b  
($) f x = f x

Here are some examples:

f g h x <=> (((f g) h) x)  
  
f g $ h x <=> f g (h x)  
f $ g h x <=> f (g h x) <=> f ((g h) x)  
f $ g $ h x <=> f (g (h x))  
  
(f . g) x <=> f . g $ x <=> f (g x)  
(f . g . h) x <=> f . g . h $ x <=> f (g (h x))

### More on the syntax

1. Infix and prefix notation

* square :: Num a => a -> a  
  square x = x ^ 2
* Any infix operator can be used in prefix notation.
* square' x = (^) x 2  
  square'' x = (^2) x
* We can remove x from the right hand side, this is called *η*-reduction.
* square''' = (^2)
* All these functions are identical.
* And functions in Haskell can be used in infix notation as well.
* add :: Num a => a -> a -> a  
  add = (+)  
    
  5 `add` 4 <=> add 5 4 <=> 9

1. Conditions

* Type class Ord is for types that can be ordered.
* absolute :: (Ord a, Num a) => a -> a  
  absolute x = if x >= 0 then x else -x
* In Haskell if statements must always have then and else.
* Here is another way to write that function:
* absolute' x  
   | x >= 0 = x  
   | otherwise = -x
* In Haskell indentation is very important. Just like in Python programs with incorrect indentation will not work or, in some cases, will work, but not the way it was intended. Haskell uses spaces instead of tabs, if you try to use tabs then the program won't compile.

### Functional style

Let's introduce a problem and then solve it using first Python and then Haskell.

We want a function that takes a list of integers and returns the sum of all even numbers in that list.

[1, 2, 3, 4, 5] -> 2 + 4 -> 6

def evenSum(l):  
 result = 0  
 for x in l:  
 if(x % 2 == 0):  
 result += x  
 return result

We can't implement it in Haskell exactly the same way because it doesn't have loops or mutable variables. So here is how we can implement it in Python without mutating variables or using loops.

def accumSum(l, n):  
 if(len(l) == 0):  
 return n  
 else:  
 x, \*xs = l  
 if(x % 2 == 0):  
 return accumSum(xs, x + n)  
 else:  
 return accumSum(xs, n)  
  
def evenSum(l):  
 return accumSum(l, 0)

Before we start, here are some Haskell functions we will use.

even :: Integral a => a -> Bool -- returns True only if the given number is even  
head :: [a] -> a -- returns the first element of the given list  
tail :: [a] -> [a] -- returns the given list without the first element

Here is our first solution:

evenSum :: [Integer] -> Integer  
evenSum l = accumSum 0 l  
  
accumSum :: Integer -> [Integer] -> Integer  
accumSum n l = if l == []  
 then n  
 else let x = head l  
 xs = tail l  
 in if even x  
 then accumSum (n+x) xs  
 else accumSum n xs

We can do several improvements to this piece of code. First we can make the type declaration more general (without changing the implementation).

evenSum :: Integral a => [a] -> a

We don't want accumSum to be a global variable, so we can make it local using where clause. Also we can use pattern matching instead of head and tail. Then we can use *η*-reduction to get this:

evenSum :: Integral a => [a] -> a  
evenSum = accumSum 0  
 where accumSum n [] = n  
 accumSum n (x:xs) = if even x  
 then accumSum (n+x) xs  
 else accumSum x xs

Pattern matching is using values instead of variable arguments. We can't use any function we want on the left side - only type constructors, which I will discuss later.

We can simplify this even more using higher order functions.

### Higher order functions

Higher order functions are functions that take another function as an argument. Here are several examples:

filter :: (a -> Bool) -> [a] -> [a]  
map :: (a -> b) -> [a] -> [b]  
foldl :: (a -> b -> a) -> a -> [b] -> a  
(.) :: (b -> c) -> (a -> b) -> a -> c  
($) :: (a -> b) -> a -> b

Function filter takes a function of type a -> Bool and a list [a]. It returns a list that only contains the elements of the given list that return True when the given function is applied.

map takes a function and a list and applies the function to every element of the list.

filter even [1..5] <=> [2, 4]  
  
map (\*2) [1..5] <=> [2,4,6,8,10]

Let's use this.

evenSum l = mysum $ filter even l  
 where mysum n [] = 0  
 mysum n (x:xs) = mysum (n+x) xs

Now, what is foldl?

foldl :: (a -> b -> a) -> a -> [b] -> a  
foldl op prev [] = prev  
foldl op prev (x:xs) = foldl op (prev `op` x) xs

foldl f z [x1,x2,x3,x4] <=> f (f (f (f z x1) x2) x3) x4

So let's use it for our problem.

evenSum :: Integral a => [a] -> a  
evenSum = foldl (+) 0 . filter even

### Defining your own types

1. type

* type TypeName = AnotherType just makes a type synonym of String.
* type Name = String
* Name and String are the same type. This is useful for making type declarations more meaningful.

1. data

* data NewDataType = TypeConstructor AnotherType is how we make a new simple type. This code makes a type constructor which is a special function that allows us to create instances of the NewDataType. We don't need to write an implementation for this function, we get it by defining the type.
* TypeConstructor :: AnotherType -> NewDataType
* Now AnotherType and NewDataType are two different types even though they represent the same data. This means that if we have a function that takes an argument of type AnotherType then it won't compile if we pass it something of type NewDataType. To extract data we can use pattern matching on type constructors.
* toOriginalType :: NewDataType -> AnotherType  
  toOriginalType (TypeConstructor thing) = thing
* Constructors can have multiple arguments or none at all. We can use the name of the type as the constructor name, which is what people usually do when there is only one constructor.
* data Thing = Thing  
    
  data StringPair = StringPair String String
* We can have types with multiple constructors.
* data MaybeString = JustString String | NoString
* This code creates a new type MaybeString with two constructors: JustString and NoString. We can do pattern matching on both of the constructors.
* hasString :: MaybeString -> Bool  
  hasString (JustString \_) = True  
  hasString NoString = False
* In pattern matching we can replace a variable with an underscore if we don't use that variable.
* data Person = Person String Int  
    
  name :: Person -> String  
  name (Person str \_) = str  
    
  age :: Person -> String  
  age (Person \_ n) = n
* Instead of writing functions name and age we can use fields and the compiler will generate them.
* data Person = Person { name :: String  
   , age :: Int  
   }
* This gives us the same name and age functions.

### Recursive types

1. Lists

* List is a common example of a recursive type. Here is how we can define the list type:
* data List a = Empty | Cons a (List a)
* Type List takes another type as an argument. We can see two constructors, here are their types:
* Empty :: List a  
  Cons :: a -> List a -> List a
* Haskell allows the use of special characters in names, this gives us the definition of lists from the standard library:
* data [] a = [] | a : [a]
* If we tried to print our new list it wouldn't work, because we don't have a function for conversion to string defined for it. Haskell has function show :: Show a => a -> String which is defined in the type class Show. So we can make our List an instance of Show. However, for predefined type classes, we can use a simpler approach. We can just derive that instance.
* data List a = Empty | Cons a (List a)  
   deriving (Show)
* We can also derive type class instances for Read (parsing strings), Eq (checking for equality), Ord (ordering), etc. This way we can get a lot of functions for free.
* data List a = Empty | Cons a (List a)  
   deriving (Show, Read, Eq, Ord)

1. Trees

* Here is another example of a recursive data type - binary trees.
* data BinTree a = Empty  
   | Node a (BinTree a) (BinTree a)  
   deriving (Show)
* Because we used an arbitrary type variable a in the type declaration we can make a lot of different trees. For example we can make trees of trees.

### Infinite structures

Haskell uses lazy evaluation, which is why we can have infinite data structures. For example in Haskell we can do this:

numbers :: [Integer]  
numbers = 1 : map (+1) numbers  
  
main = print $ take 3 numbers

The function take takes the first n numbers from the given list. If we run this code it won't get stuck in an infinite recursion, it will print [1,2,3]. Because of lazy evaluation Haskell doesn't calculate all the numbers in the list, but only the ones that it needs.

In this example we just have all positive integers. Let's take a look at a more interesting example with a tree.

tree :: BinTree Integer  
tree = Node 0 (dec tree) (inc tree)  
 where dec (Node x l r) = Node (x-1) (dec l) (dec r)  
 inc (Node x l r) = Node (x+1) (inc l) (inc r)

|(-2)..  
 |(-1)-|  
 | |( 0)..  
0-|  
 | |( 0)..  
 |( 1)-|  
 |( 2)..

### Functors

Functor is one of the most important abstractions in Haskell. Basically, it is a type class that generalizes the map function.

class Functor f where  
 fmap :: (a -> b) -> f a -> f b

The notion of functors comes from maths, and in maths there are laws for it. Unfortunately GHC doesn't support laws in type classes, so it's programmers' responsibility to make sure they work. The only relevant to Haskell law is that if we have two functions: h :: a -> b and f :: b -> c then for any functor fmap (f . h) should be the same as fmap f . fmap h. <$> is a infix operator for fmap.

f <$> x = fmap f x

Here are some examples of functors:

data Maybe a = Just a | Nothing  
  
instance Functor Maybe where  
 fmap f (Just x) = Just $ f x  
 fmap \_ Nothing = Nothing  
  
maybeFive :: Maybe Int  
maybeFive = Just 5  
  
maybeSix :: Maybe Int  
maybeSix = fmap (+1) maybeFive -- = Just 6  
  
data [] a = [] | a : [a]  
  
instance Functor [] where  
 fmap f (x:xs) = f x : fmap f xs  
 fmap \_ [] = []  
 -- fmap = map  
  
data Either a b = Left a | Right b  
  
instance Functor (Either a) where  
 fmap f (Right x) = Right $ f x  
 fmap \_ (Left x) = Left x  
  
numberOrString :: Either Int String  
numberOrString = Right "World"  
  
numberOrHello :: Either Int String  
numberOrHello = ("Hello " ++) <$> numberOrString -- Right "Hello World"  
  
numOrStr :: Either Int String  
numOrStr = Left 5  
  
numOrHello :: Either Int String  
numOrHello = ("Hello " ++) <$> numOrHello -- Left 5  
  
data (,) a b = (,) a b  
  
instance Functor ((,) a) where  
 fmap f (x, y) = (x, f y)  
  
pairOfNumbers :: (Int, Int)  
pairOfNumbers = (2, 3)  
  
incrementedPair :: (Int, Int)  
incrementedPair = fmap (+1) pairOfNumbers -- = (2, 4)

### Applicative functors

As you know Maybe is a functor. This is why we can do this:

Prelude> negate <$> Just 2  
Just (-2)

But what if we want to add two Maybe numbers.

Prelude> :t (+) <$> Just 2  
(+) <$> Just 2 :: Num a => Maybe (a -> a)

After we partially apply addition using fmap we get a function inside a functor. How to apply that function to our second Maybe number? Use applicative functors.

class Functor f => Applicative f where  
 pure :: a -> f a  
 <\*> :: f (a -> b) -> f a -> f b

Maybe is an applicative functor, hence we can do this:

Prelude> (+) <$> Just 2 <\*> Just 3  
Just 5

Applicative functors also have laws:

pure id <\*> v <=> v -- identity  
pure f <\*> pure x <=> pure (f x) -- homomorphism  
u <\*> pure y <=> pure ($ y) <\*> u -- interchange  
pure (.) <\*> u <\*> b <\*> w <=> u <\*> (v <\*> w) -- composition

Here are some examples of applicative functors:

data Maybe a = Just a | Nothing  
  
instance Applicative Maybe where  
 pure = Just  
 (Just f) <\*> (Just x) = Just $ f x  
 \_ <\*> \_ = Nothing  
  
data [] a = [] | a : [a]  
  
instance Applicative [] where  
 pure x = [x]  
 \_ <\*> [] = []  
 [] <\*> \_ = []  
 (f:fs) <\*> l = (f <$> l) ++ (fs <\*> l)  
 -- applied every function to every element of the list  
  
data Reader r a = Reader { runReader :: r -> a }  
  
instance Applicative (Reader r) where  
 pure g = Reader $ const g -- const :: a -> b -> a  
 f <\*> g = Reader $ \r -> runReader f r $ runReader g r

### Monads

headMay :: [a] -> Maybe a  
headMay [] = Nothing  
headMay (x:\_) = Just x

Assume we have a list of lists and we want to safely get the first element of the first list. We can't use head as it will crash if you call it with an empty list, so we need to apply headMay twice. We can try using fmap headMay . headMay, but then we'll get this:

Prelude> :t fmap headMay . headMay  
fmap headMay . headMay :: [[a]] -> Maybe (Maybe a)

We want to reduce Maybe (Maybe a) to just Maybe a. Another example is if we want to convert a list of lists into a single list. Both of these problems can be solved using monads. Here are some definitions:

const :: a -> b -> a  
const x \_ = x  
  
class Applicative m => Monad m where  
 (>>=) :: m a -> (a -> m b) -> m b  
 (>>) :: m a -> m b -> m b  
 x >> y = x >>= const y -- default implementation  
  
instance Monad Maybe where  
 (Just x) >>= f = f x  
 Nothing >>= \_ = Nothing  
  
instance Monad [] where  
 (x:xs) >>= f = f x ++ (xs >>= f)  
 [] >>= \_ = []

Now for the first problem we can do this:

headMay l >>= headMay

l is the list of lists. And here is how we can solve the second problem:

Prelude> :t (>>= id)  
(>>= id) :: Monad m => m (m b) -> m b  
Prelude> [[1..5],[6..10]] >>= id  
[1,2,3,4,5,6,7,8,9,10]

If we import Control.Monad we'll get several helper functions for working with monads.

join :: m (m a) -> m a  
join = (>>= id)  
  
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)  
(>=>) f h = \x -> f x >>= h

Prelude> headMay l = if length l == 0 then Nothing else Just $ head l  
Prelude> import Control.Monad  
Prelude Control.Monad> :t join  
join :: Monad m => m (m a) -> m a  
Prelude Control.Monad> join [[1..5],[6..10]]  
[1,2,3,4,5,6,7,8,9,10]  
Prelude Control.Monad> :t headMay >=> headMay  
headMay >=> headMay :: [[c]] -> Maybe c

### IO

In Haskell functions are pure, however printing to console, reading/writing files, and other IO actions don't give the same results every time you call them. To deal with IO actions Haskell has a special monad - IO monad. This allows us to isolate pure and impure parts of the code. In our program we have main procedure which has type IO ().

data () = ()

1. Printing to console

* putStr :: String -> IO () -- prints the given string   
  putStrLn :: String -> IO () -- prints the given string and starts a new line  
  print :: Show a => a -> IO ()  
  print = putStrLn . show
* Now we can write a "Hello World" program.
* main :: IO ()  
  main = print "Hello World!"

1. Reading user console input

* getChar :: IO Char  
  getLine :: IO String
* Notice that these are not functions, they are IO actions. Now we can write a program that asks for the user's name and prints "Hello <username>!".
* main :: IO ()  
  main = print "What's your name?" >> getLine >>= print . ("Hello " ++) . (++ "!")

1. Do notation

* We can use a simpler notation for monads that is more similar to imperative programming languages.
* main :: IO ()  
  main = do print "What's your name?"  
   name <- getLine  
   print $ "Hello " ++ name ++ "!"
* In this case every line must be an IO action. This syntax is a nicer way of writing this:
* main :: IO ()  
  main = print "What's your name?"  
   >> getLine  
   >>= \name -> print ("Hello " ++ name ++ "!")
* For the compiler these two things are identical. We can use do notation not only with the IO monad, but with any monad.
* headMay :: [a] -> Maybe a  
  headMay (x:xs) = Just x  
  headMay [] = Nothing  
    
  headOfHead :: [[a]] -> Maybe a  
  headOfHead l = do h <- headMay l  
   headMay h

## Spec for the examples

To show that Haskell is better than other high-level programming languages I will solve several problems in Haskell and Ruby. Ruby is a high-level programming language, and it is almost the exact opposite of Haskell: it's imperative (although it supports some features from functional programming, as Haskell is one of the languages that Ruby was inspired by, I will avoid using them to show more differences between imperative and functional programming), dynamically typed, interpreted, and object oriented.

### Example 1: sorting a big file

In this example I will write a script that reads numbers from a file, sorts them, and writes the sorted list to another file. Even though this investigation is about high-level languages, I decided to include a solution to this problem in a low-level language C. I did this to make a more representative performance comparison. In order to show this I will measure time taken for each of the scripts to process a file with one million random integers.

### Example 2: reverse polish server

In this example I will implement a client-server system. The client takes an expression in reverse polish notation and an action (check or evaluate), then the expression gets sent to the server where the required action gets executed, finally the client shows the result of performing the given action on the given expression.

By comparing methods for defining and implementing an API this example is to show Haskell's safety, expressiveness, high-level abstractions, and modularity.

# Design

## Sorting a big file

The script needs to read the file "random\_numbers", which contains comma-separated integers, parse the contents to get the list of integers, sort them, convert back to the original format, and write the result to the file "{language}\_result" (where {language} is the name of the language that was used for the script).

## Reverse polish server

In the script there needs be a data type that represents a simple mathematical expression (in terms of numbers and operators +, -, \*, /). For this data type a function that evaluates the expression must be defined. If the function is called on a number then this number gets returned, if the function is called on an expression then the function gets recursively called on the operands and the current operator is applied to the two results.

There must be a function that takes a string and reads the expression in reverse polish notation that is stored in it. The function returns a nullable expression of the expression type. The function uses a stack of expressions. When it sees an operator in the input it takes two expressions from the stack and constructs a new expression with the operator it read as the operator and the two expressions as the operands and puts the new expression in the stack. The function treats the rest of the input as numbers delimited by spaces. After the function finishes going through the entire input string if there is only one element in the stack then it returns it, otherwise it return a null value because the expression is invalid.

These functions are then used to implement the following API endpoints:

* POST: /check - extracts an expression in reverse polish notation in JSON format from the request body and returns a response with a boolean in JSON format. If the given expression is valid then the server responds with true, if the given expression is invalid then the server responds with false.
* POST: /evaluate - extracts an expression in reverse polish notation in JSON format from the request body and returns a response with code 200 and response body containing the result of evaluating the expression if the expression is valid, otherwise returns a response with error code 400 (Bad Request) and error message "invalid".

# Solution

## Sorting a big file

### Code

1. Haskell

* {-# LANGUAGE OverloadedStrings #-}  
    
  import qualified Data.ByteString.Lazy.Char8 as C  
  import Data.List (sort)  
    
  main :: IO ()  
  main = C.readFile "random\_numbers"  
   >>= maybe (print "Failed to parse!")  
   ( C.writeFile "haskell\_result"  
   . C.intercalate "," . fmap (C.pack . show) . sort . fmap fst  
   ) . traverse C.readInt . C.split ','
* The first line enables a language extension called OverloadedStrings. It allows us to use different types as strings. For example, in this script "random\_numbers" is a standard string and "," is a byte string. The compiler can infer the right type of string from type definitions - the first argument of C.readFile is of type String and the first argument of C.intercalate is ByteString.
* Then I imported two modules. The first one is from a library called bytestring. The default Haskell strings are very inefficient as they are just lists of characters, but there are different alternatives. One of them is using byte strings, which are arrays of bytes. There are two kinds of byte strings: strict and lazy. In this case I used a special version of lazy byte strings that interprets each byte as a character. The keyword qualified in the import statement means that the contents of the module won't be in the global namespace. as C means that we refer to the module as C. For example, we can write C.pack instead of Data.ByteString.Lazy.Char8.pack.
* Secondly I imported sort function from the Data.List module. It's an implementation of the merge sort algorithm. One of classical examples of Haskell code, that shows how nice and expressive it is, is the Quicksort function.
* qsort :: Ord a => [a] -> [a]  
  qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)  
  qsort [] = []
* At first glance it looks similar to the quicksort algorithm, but it's actually less efficient. It uses the same idea - divide and conquer, however the performance of the original quicksort function relies on the very fast swap mechanism, which is not something we can easily do in Haskell. As Haskell uses immutable data structures it doesn't swap any values in memory, it creates new ones. This is why merge sort is usually more efficient than quicksort in Haskell.
* In main I have a composition of many different functions. Let's quickly take a look at every one of them.
* C.readFile :: FilePath -> IO C.ByteString
* FilePath is a type synonym for String. C.readFile takes a file path and returns the contents of the file as a byte string.
* C.split :: Char -> C.ByteString -> [C.ByteString]
* This function breaks a byte string into pieces separated by the first argument, consuming the delimiter.
* C.readInt :: C.ByteString -> Maybe (Int, C.ByteString)
* C.readInt reads an Int from the beginning of the given byte string. If it fails to do that then it returns Nothing, otherwise it returns the integer and the rest of the string.
* class Foldable t where  
   foldr :: (a -> b -> b) -> b -> t a -> b  
    
  class (Functor t, Foldable t) => Traversable t where  
   traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
* traverse maps each element of a structure to an action, evaluates these actions from left to right, and collects the result.
* traverse C.readInt :: Traversable t => t C.ByteString -> Maybe (t (Int, C.ByteString))
* List is in the Traversable type class, which is why we can compose this with C.split ','.
* traverse C.readInt . C.split ',' :: C.ByteString -> Maybe [(Int, C.ByteString)]
* maybe :: b -> (a -> b) -> Maybe a -> b
* The type fully explains what the function does.
* C.pack :: [Char] -> C.ByteString
* C.pack takes a string and converts it into a byte string.
* C.intercalate :: C.ByteString -> [C.ByteString] -> C.ByteString
* C.intercalate joins a list of byte strings, putting the first argument between each element of the list.
* fmap fst :: Functor f => f (b1, b2) -> f b1  
    
  sort . fmap fst :: Ord a => [(a, b)] -> [a]  
    
  C.pack . show :: Show a => a -> C.ByteString  
    
  fmap (C.pack . show) . sort . fmap fst :: (Ord a, Show a) => [(a, b)] -> [C.ByteString]  
    
  C.intercalate "," . fmap (C.pack . show) . sort . fmap fst  
   :: (Ord a, Show a) => [(a, b)] -> C.ByteString
* C.writeFile :: FilePath -> C.ByteString -> IO ()
* C.writeFile takes a file path and a byte string and writes the byte string to the file, overwriting existing data or creating the file if it doesn't exist.
* C.readFile "random\_numbers" :: IO C.ByteString  
    
  maybe (print "Failed to parse!")  
   ( C.writeFile "haskell\_result"  
   . C.intercalate "," . fmap (C.pack . show) . sort . fmap fst  
   ) . traverse C.readInt . C.split ','  
   :: C.ByteString -> IO ()  
    
  (>>=) :: Monad m => m a -> (a -> m b) -> m b
* If we put all these things together we'll get main. In summary, it reads numbers from "random\_numbers", splits the string with comma separated integers into a list of byte strings with integers, then parses each integer, prints "Failed to parse!" in case it fails to parse, otherwise sorts the list of integers, converts each integer back into a byte string, joins the byte strings and writes the result to "haskell\_result".

1. Ruby

* input\_file\_name = 'random\_numbers'  
  output\_file\_name = 'ruby\_result'  
    
  buffer = ''  
  numbers = []  
    
  # open the input file  
  File.open(input\_file\_name) do |f|  
   # for each character c in the file  
   f.each\_char do |c|  
   if c == ','  
   # convert the buffer to an integer and add to the list of numbers  
   numbers << Integer(buffer)  
   # empty the buffer  
   buffer = ''  
   else  
   # add the character to the buffer  
   buffer << c  
   end  
   end  
    
   # convert the buffer to an integer and add to the list of numbers  
   numbers << Integer(buffer)  
  end  
    
  # sort the numbers  
  numbers = numbers.sort  
    
  # open the output file  
  File.open(output\_file\_name, 'w') do |f|  
   # remove the last number from the list  
   last = numbers.pop  
   # write all the remaining numbers with a comma after each of them to the output file  
   numbers.each { |num| f.write "#{num}," }  
   # write the last element  
   f.write last  
  end

1. C

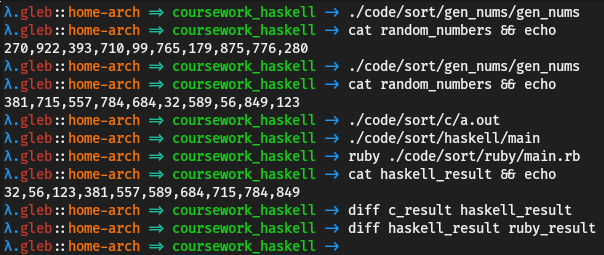
* #define SIZE (1000000)  
  #define INPUT\_FILE ("random\_numbers")  
  #define OUTPUT\_FILE ("c\_result")  
    
  #include <stdio.h>  
  #include <stdlib.h>  
  #include <string.h>  
    
  // Just difference of two numbers  
  int cmpfunc(const void \* a, const void \* b)  
  {  
   return (\*(int\*)a - \*(int\*)b);  
  }  
    
  int main()  
  {  
   // Initializing the file pointer  
   FILE \*fs;  
    
   // current char and buffer for digits  
   char ch, buffer[32];  
   int i = 0, arr[SIZE], j = 0;  
    
   // Openning the file with file handler as fs  
   fs = fopen(INPUT\_FILE, "r");  
    
   // Read the file unless the file encounters an EOF  
   for(ch = fgetc(fs); ; ch = fgetc(fs)) {  
   if(ch == ',') {  
   // Converting the content of the buffer into an array position  
   arr[j] = atoi(buffer);  
    
   // Increamenting the array position  
   j++;  
    
   // Clearing the buffer, this function takes two  
   // arguments, one is a character pointer and  
   // the other one is the size of the character array  
   bzero(buffer, 32);  
    
   // setting the buffer index to 0  
   i = 0;  
   }  
   else if (ch != EOF) {  
   // add the next character to the buffer  
   buffer[i] = ch;  
   // increment the buffer index  
   i++;  
   }  
   else { // end of the file  
   // add the number from the buffer to  
   arr[j] = atoi(buffer);  
    
   // end the loop  
   break;  
   }  
   }  
    
   // close the file  
   fclose(fs);  
    
   // sort the array  
   qsort(arr, SIZE, sizeof(int), cmpfunc);  
    
   // open the output file  
   fs = fopen(OUTPUT\_FILE, "w");  
    
   // write every number (except the last one) with a comma after each  
   for(i = 0; i < SIZE - 1; i++) {  
   fprintf(fs, "%d,", arr[i]);  
   }  
    
   // write the last number  
   fprintf(fs, "%d", arr[i]);  
    
   // close the file  
   fclose(fs);  
    
   // return 0 (success code)  
   return 0;  
  }

### Tests

I wrote a script that generates a list of random numbers in range and writes them to a file separated by commas to test the sorting script.

import Control.Monad  
import System.Random  
  
numOfNums :: Integer  
numOfNums = 10  
  
file :: FilePath  
file = "random\_numbers"  
  
main :: IO ()  
main = join  
 $ (\(r:rs) -> foldl (\p x -> p >> addToFile (',' : show x)) (writeFile file $ show r) rs)  
 <$> foldl (\rs \_ -> (:) <$> (randomRIO (1, 1000) :: IO Int) <\*> rs) (return []) [1..numOfNums]  
 where addToFile = appendFile file

I changed SIZE macro in the C script to 10 for the purpose of the test.



### Evaluation

1. Safety

* Let's take a look at the function C.readInt. It returns Maybe (Int, C.ByteString). In most languages you can work with nullable types without checking if they are actually null, but Haskell doesn't allow that. It forces you to do something with the fact that a value can be Nothing. In this case I covered the case when it's Nothing by using the function maybe and providing the default behavior for that situation. If you want you can unsafely cast Maybe a to a using the function fromJust from the Data.Maybe module. However, the compiler won't make that decision for you and you'll have to explicitly tell it to do so. This shows how Haskell is a safer language.

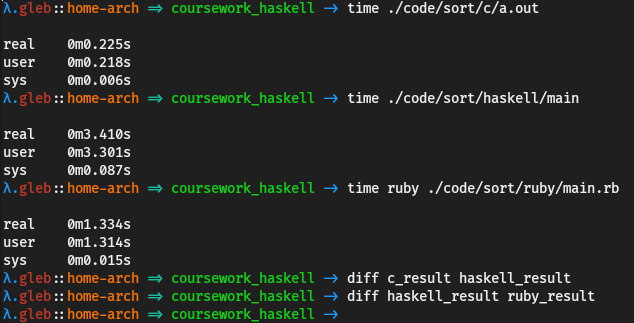
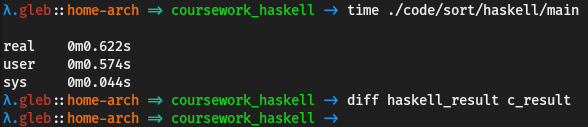
1. Expressiveness

* As you can see we didn't need a lot of code to solve the problem. If you take a look at the way the algorithm was described in English in <sec:design_sorting> you'll see that the code I wrote does exactly that. We basically tell Haskell what we want to achieve and not how to achieve it.
* {-# LANGUAGE OverloadedStrings #-}  
    
  import qualified Data.ByteString.Lazy.Char8 as C  
  import Data.List (sort)  
    
  main :: IO ()  
  main = C.readFile "random\_numbers" -- read the file "random\_numbers"  
   >>= maybe (print "Failed to parse!")  
   ( C.writeFile "haskell\_result" -- write the result to the file "haskell\_result"  
   . C.intercalate "," . fmap (C.pack . show) -- convert back to the original format  
   . sort -- sort them  
   . fmap fst ) . traverse C.readInt . C.split ',' -- parse comma-separated integers
* In the Ruby solution, as you can see in the source code, the code represents a sequence of instructions which the computer needs to do. The Haskell version of the program has less code in it (even if we remove the comments) and the structure of the Haskell script is closer to the way the problem was defined in English, which shows us the expressiveness of the language.

1. Modularity

* This script also shows how modular Haskell is. To solve the problem I just glue together 13 different functions using 2 operators. If we want to reuse some of the functionality we can easily extract the piece of code that does what we want from main and put it in another function. For example, let's say we want to reuse the code for parsing.
* {-# LANGUAGE OverloadedStrings #-}  
    
  import qualified Data.ByteString.Lazy.Char8 as C  
  import Data.List (sort)  
    
  parse :: C.ByteString -> Maybe [Int]  
  parse = fmap (fmap fst) . traverse C.readInt . C.split ','  
    
  main :: IO ()  
  main = C.readFile "random\_numbers"  
   >>= maybe (print "Failed to parse!")  
   ( C.writeFile "haskell\_result"  
   . C.intercalate "," . fmap (C.pack . show) . sort  
   ) . parse
* As you can see, in Haskell it's very easy to compose and decompose code.

1. Performance

* To test performance I generated a file with one million random numbers (I changed macro SIZE in the C script back to a million) and measured the execution time.
* 
* As you can see Haskell didn't perform very well in this test. Why is that? I used sort function that applies mergesort algorithm on immutable lists. This is a problem for performance for several reasons: mergesort is not very fast, lists are not very fast, we need to allocate memory very often.
* I solved this problem by replacing the sort function. I used unboxed vectors (using vector library), safe (internal) mutations, and introspective sorting (using vector-algorithms library).
* {-# LANGUAGE OverloadedStrings #-}  
    
  import qualified Data.ByteString.Lazy.Char8 as C  
  import qualified Data.Vector.Algorithms.Intro as Alg  
  import qualified Data.Vector.Unboxed as U  
    
  sort :: (Ord a, U.Unbox a) => [a] -> [a]  
  sort = U.toList . U.modify Alg.sort . U.fromList  
    
  main :: IO ()  
  main = C.readFile "random\_numbers"  
   >>= maybe (print "Failed to parse!")  
   ( C.writeFile "haskell\_result"  
   . C.intercalate "," . fmap (C.pack . show) . sort . fmap fst  
   ) . traverse C.readInt . C.split ','
  1. Unboxed vectors
  + vector library provides efficient arrays. Unboxed types are raw values. So boxed (default) vectors are arrays of pointers and unboxed vectors are arrays of raw values.
  + U.fromList :: U.Unbox a => [a] -> U.Vector a  
    U.toList :: U.Unbox a => U.Vector a -> [a]
  + As you can guess from the names and types U.fromList converts a list of values that can be represented as raw values to an unboxed vector and U.toList converts an unboxed vector to a list.
  1. Introsort
  + Introspective sorting or introsort is an optimised version of quicksort. From the description of the module Data.Vector.Algorithms.Intro:
  + This module implements various algorithms based on the introsort algorithm, originally described by David R. Musser in the paper *Introspective Sorting and Selection Algorithms*. It is also in widespread practical use, as the standard unstable sort used in the C++ Standard Template Library.
  + Introsort is at its core a quicksort. The version implemented here has the following optimizations that make it perform better in practice:
    - Small segments of the array are left unsorted until a final insertion sort pass. This is faster than recursing all the way down to one-element arrays.
    - The pivot for segment [l,u) is chosen as the median of the elements at l, u-1 and (u+l)/2. This yields good behavior on mostly sorted (or reverse-sorted) arrays.
    - The algorithm tracks its recursion depth, and if it decides it is taking too long (depth greater than 2 \* lg n), it switches to a heap sort to maintain O(n lg n) worst case behavior. (This is what makes the algorithm introsort).
  1. Safe internal mutations
  + Let's take a look at types of U.modify and Alg.sort.
  + U.modify  
     :: U.Unbox a =>  
     (forall s. U.MVector s a -> GHC.ST.ST s ())  
     -> U.Vector a -> U.Vector a
  + First let's take a look at ST (state thread). ST is a monad, it can be described as a restricted IO monad or a monad for pure mutations. Some functions are more efficient with mutable memory, but global mutable memory is unsafe. This is why we have the ST monad. With ST you can use internal mutations, but the whole computation "thread" is not allowed to exchange mutable state with the outside world. Using this monad you can make functions that take in normal Haskell values, then allocate mutable memory, work with it, and return normal Haskell values back.
  + ST type takes two types as arguments. The first argument is the scope. This is how we can be sure that the computation is pure. If the first argument is an arbitrary type variable then we know that the computation doesn't depend on the initial state, hence it is pure. The second argument is the output state. It is worth mentioning that ST provides **strict** state threads.
  + U.MVector s a is a mutable vector of type a in scope s.
  + forall s. means that s can be anything. In this case it's used not to make U.modify parametrically polymorphic in c, but to make sure that the function passed as an argument is parametrically polymorphic in c. This is done so that the scope of ST of the result type of the argument function has arbitrary type. In other words, this way we can be sure that the given function returns a pure computation.
  + So U.modify takes a function that does a pure computation in ST and an unboxed vector, and it returns a new vector which is the result of applying the given computation to the given vector.
  + Alg.sort  
     :: (Ord e, Data.Vector.Generic.Mutable.Base.MVector v e,  
     Control.Monad.Primitive.PrimMonad m) =>  
     v (Control.Monad.Primitive.PrimState m) e -> m ()
  + Data.Vector.Generic.Mutable.Base.MVector is a class of mutable vectors and U.MVector is in it.
  + PrimMonad is a type class for primitive state-transformer monads (IO and ST). IO and ST have many operations that are almost the same for both of the monads, which is why PrimMonad type class was created. This means that Alg.sort works with both ST and IO. PrimState is defined in the type class PrimMonad. It's an associated type giving the type of the state token (s in case of ST s).
  + Alg.sort takes a mutable vector and sorts it, returning the unit type () wrapped in a state-transformer monad. So we can pass Alg.sort as an argument to U.modify.
  + U.modify Alg.sort :: (Ord a, U.Unbox a) => U.Vector a -> U.Vector a
  1. The result of the optimizations
  + 
  + As you can see this significantly improved performance. If this still isn't fast enough for you, there are other optimizations that can be done: you can use the foreign function interface to call C functions, reduce the number of different conversions in the script, completely get rid of lists, etc. This shows that Haskell can have decent performance. Depending on what application you're developing you can optimize Haskell to get the performance you need. It's still slower than low-level languages like C, but if you really need certain parts of your code to perform really well then you can use foreign function interface and call C code from Haskell.

1. Abstractions

* This example also shows what Haskell's high level abstractions can do. A good example of the language's use of high-level abstractions from the optimized version of the script is how Haskell uses type system to ensure that a state mutating computation is pure using ST monad. The type system plays a big role in Haskell. Haskell's type system is very strict, but at the same time it uses type variables and type classes, making the language very flexible and allowing you to define the most general versions of functions, variables, etc.

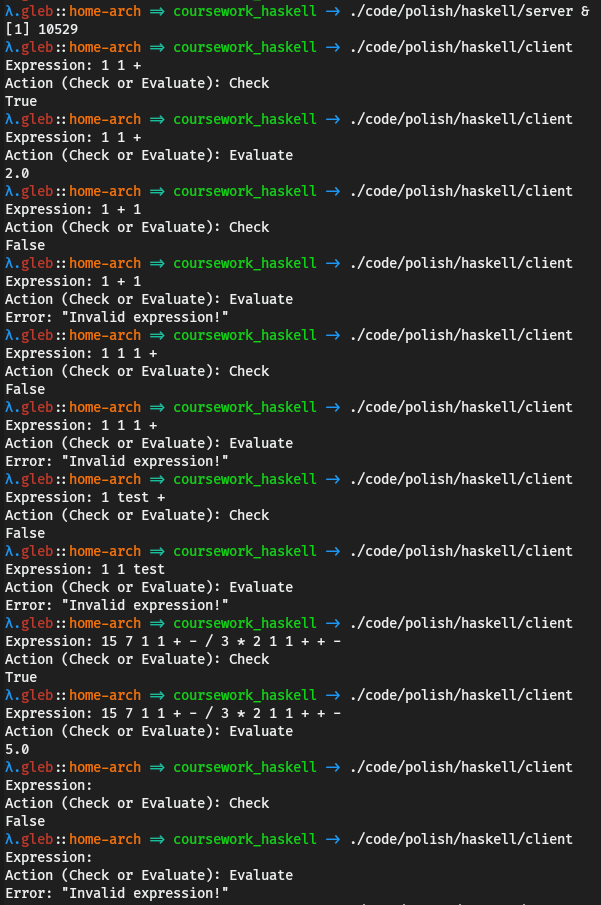
## Reverse Polish server

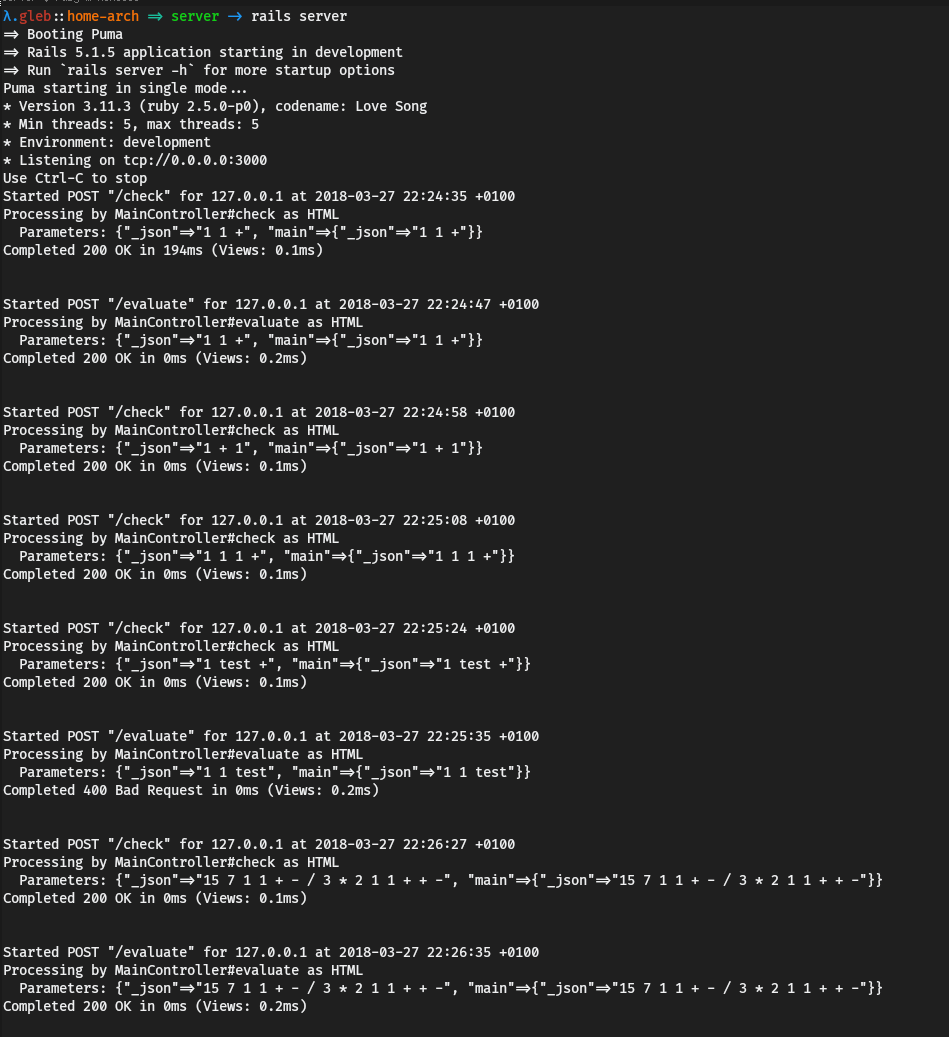
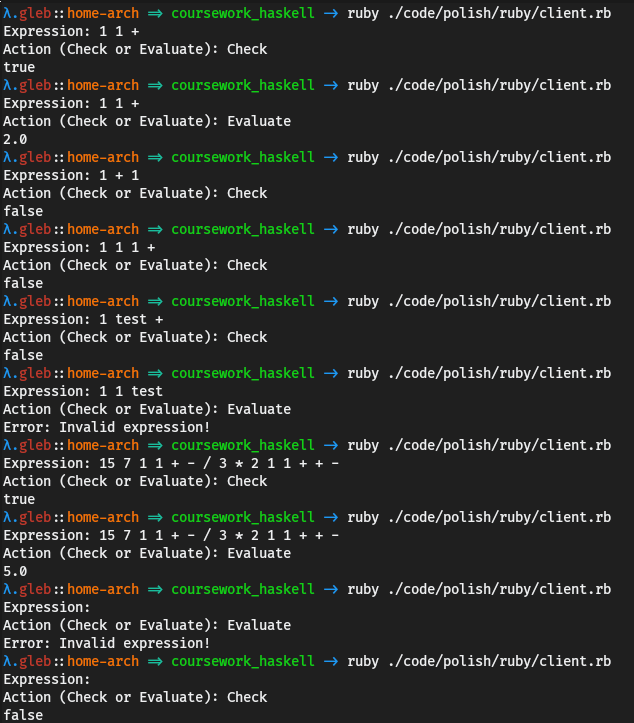
### Code

1. Haskell
   1. Server
   * To implement the server I created two files: one for the API and one for the implementation of the API. First I defined the API:
   * {-# LANGUAGE DataKinds #-}  
     {-# LANGUAGE TypeOperators #-}  
       
     module API where  
       
     import Data.Proxy  
     import Servant  
       
     type API = "check" :> ReqBody '[JSON] String :> Post '[JSON] Bool  
      :<|> "evaluate" :> ReqBody '[JSON] String :> Post '[JSON] Float  
       
     api :: Proxy API  
     api = Proxy
   * In the beginning of the script I enable two language extensions. DataKinds language extensions promotes values to types. The same way as values have types, types have kinds.
   * Prelude> :set -XDataKinds -- this is how you enable language extensions in ghci  
     Prelude> :t 5  
     5 :: Num p => p  
     Prelude> :t (+)  
     (+) :: Num a => a -> a -> a  
     Prelude> :k Int  
     Int :: \*  
     Prelude> :k Either  
     Either :: \* -> \* -> \*
   * DataKinds allows us to use values as types and types as kinds (we still can use them the regular way though).
   * Prelude> :set -XDataKinds  
     Prelude> data Response = Response  
     Prelude> :t Response  
     Response :: Response  
     Prelude> :k 'Response  
     'Response :: Response
   * TypeOperators lets us define a type as an operator.
   * {-# LANGUAGE TypeOperators #-}  
       
     data path :> a  
     data l :<|> r = l :<|> r
   * Notice that :> doesn't have any type constructors. This means that there are no values of this type, but we can still use this type operator for type-level computations.
   * Prelude> :k (:>)  
     (:>) :: \* -> \* -> \*  
     Prelude> data l :<|> r = l :<|> r  
     Prelude> :t (:<|>)  
     (:<|>) :: l -> r -> l :<|> r  
     Prelude> :k (:<|>)  
     (:<|>) :: \* -> \* -> \*
   * These type operators are defined in a library called Servant, which I used to implement the server and the client. Using this library you can define your API in terms of types. Using these type operators and other types provided by Servant we can define the API. Here are the types that I used:
     + Post is a type that represents a post request. It takes a type-level list of content types (response formats, like JSON or XML) and the type of the response (this type must be in the type classes that convert values of this type to formats from the content type type-level list).
     + JSON is a content type. It doesn't have a constructor and it only exists for representing the content type at the type level.
     + ReqBody is a data type that takes a type-level list of content types (request formats that it can accept) and the type of the value that is encoded in one of the content types.
   * So the type API represents the API defined in the design section (<sec:design_reverse_polish>).
     + POST: /check - takes a string in JSON format, returns a boolean in JSON format
     + POST: /evaluate - takes a string in JSON format, returns a float in JSON format
   * Of course this type only defines the endpoints, the actual server logic goes into the implementation. API is a type and sometimes we need to use the API definition as a value. In Haskell we can't pass types as arguments to functions, which is why we need Proxy. Here is how it's defined:
   * {-# LANGUAGE KindSignatures #-}  
     {-# LANGUAGE PolyKinds #-}  
       
     data Proxy (t :: k) = Proxy
   * KindSignatures extension enables explicit kind declarations and PolyKinds enables kind polymorphism.
   * Prelude> data Proxy t = Proxy  
     Prelude> :k Proxy  
     Proxy :: \* -> \*  
     Prelude> :set -XPolyKinds  
     Prelude> :set -XKindSignatures   
     Prelude> data Proxy (t :: k) = Proxy  
     Prelude> :k Proxy  
     Proxy :: k -> \*
   * This allows us to pass a value, which is always Proxy, and pass a type by explicitly stating the type of the value that we pass.
   * Now let's take a look at the implementation of the API.
   * {-# LANGUAGE DataKinds #-}  
     {-# LANGUAGE OverloadedStrings #-}  
     {-# LANGUAGE TypeOperators #-}  
       
     import Data.Proxy  
     import Network.Wai  
     import Network.Wai.Handler.Warp  
     import Safe  
     import Servant  
       
     import qualified Data.Text.IO as TIO  
     import qualified Lackey  
       
     import qualified API  
       
     infixl 6 :+:  
     infixl 6 :-:  
     infixl 7 :\*:  
     infixl 7 :/:  
       
     data Expr a = Expr a :+: Expr a  
      | Expr a :-: Expr a  
      | Expr a :\*: Expr a  
      | Expr a :/: Expr a  
      | Number a  
      deriving Show  
       
     eval :: (Num a, Fractional a) => Expr a -> a  
     eval (x :+: y) = eval x + eval y  
     eval (x :-: y) = eval x - eval y  
     eval (x :\*: y) = eval x \* eval y  
     eval (x :/: y) = eval x / eval y  
     eval (Number x) = x  
       
     parse :: (Read a, Fractional a) => String -> Maybe (Expr a)  
     parse = flip parseAccum [] . words  
      where parseAccum :: (Read a, Num a) => [String] -> [Expr a] -> Maybe (Expr a)  
      parseAccum [] [x] = Just x  
      parseAccum ("+":cs) (x1:x2:xs) = parseAccum cs $ x1 :+: x2 : xs  
      parseAccum ("-":cs) (x1:x2:xs) = parseAccum cs $ x1 :-: x2 : xs  
      parseAccum ("\*":cs) (x1:x2:xs) = parseAccum cs $ x1 :\*: x2 : xs  
      parseAccum ("/":cs) (x1:x2:xs) = parseAccum cs $ x1 :/: x2 : xs  
      parseAccum (str:cs) exprs = readMay str >>= parseAccum cs . (: exprs) . Number  
      parseAccum \_ \_ = Nothing  
       
     server :: Server API.API  
     server = maybe (return False) (const $ return True) . parse  
      :<|> maybe invalid return . fmap eval . parse  
      where invalid = throwError err400 { errBody = "Invalid expression!" }  
       
     app :: Application  
     app = serve API.api server  
       
     main :: IO ()  
     main = run 3000 app
   * First I defined 4 operators which I then used as type constructors. infixl assigns priority to an operator. I used the same priority for :+:, :-:, :\*:, :/: as Haskell uses by default for +, -, \*, /, respectively.
   * Then I defined a data type (Expr) that represents a simple mathematical expression. It's a recursive data type and it's either a number or addition/subtraction/multiplication/division of two expressions. Function eval evaluates Expr.
   * parse takes a string with an expression in reverse polish notation and parses it to get an expression of type Expr. It uses several functions for that:
   * flip :: (a -> b -> c) -> b -> a -> c -- defined in Prelude  
       
     words :: String -> [String] -- defined in Prelude  
     -- breaks a string up into a list of words, which were delimited by white space  
       
     readMay :: Read a => String -> Maybe a -- defined in Safe (library `safe`)  
     -- parses a string, returns Nothing if fails
   * To implement parse I wrote a simple local recursive function parseAccum. It takes a list of strings with terms of the given expression in reverse polish notation and a list of expressions of type Expr a, which is used as a stack. If the list of strings is empty and there is only one element in the stack then it means that we successfully parsed the given expression, so parseAccum just returns the expression from the stack. If the current element of the list of strings is an operator then parseAccum takes two top elements from the stack, constructs a new expression with the given operator, puts the new expression in the stack, and recursively calls parseAccum on the rest of the list and the new stack. If the current element of the list is not an operator then it must be a number, so parseAccum attempts to read a number from the list and if it succeeds then it puts the number into the stack and calls parseAccum on the rest of the list and the new stack. In any other case it returns Nothing.
   * server is the implementation of API. Server is a type family, which is like a type-level function. This way it can figure out what type the implementation of the API should have for any API type. The API I defined has two endpoints, both of which take data from request body, which is why the actual type is
   * server :: ([Char] -> Handler Bool) :<|> ([Char] -> Handler Float)
   * Handler is a monad from Servant. In this case I don't have any impure computations in the implementation of the server, so I just used return to get the result that matches the type definition. The implementations for /check and /evaluate are separated by :<|>. The function for /check attempts to parse the given expression and then converts Maybe (Expr Double) (unspecified arbitrary numbers from Fractional default to Double) to Handler Bool using functions maybe, const, and return. The function for /evaluate takes an expression, attempts to parse it, evaluates it and returns in the right type if parse didn't return Nothing, otherwise it sends a response with HTTP error 400 (Bad request) and message "Invalid expression!".
   * app converts the API type and the implementation of the API to Application, which is a type defined in Network.Wai (from library wai - web application interface). We need to do this because Servant doesn't provide any functions for running the server, it allows you to plug your Servant code into different web servers. WAI provides a common protocol for communication between web applications and web servers. Now that we have a WAI web application we can run it using run function from Network.Wai.Handler.Warp (from library warp), which is a web server for WAI applications. main IO action runs the application on port 3000.
   1. Client
   * {-# LANGUAGE DataKinds #-}  
     {-# LANGUAGE TypeOperators #-}  
       
     import Data.Proxy  
     import Network.HTTP.Client  
     import Safe  
     import Servant.API  
     import Servant.Client as SC  
     import System.IO  
       
     import API (api)  
       
     check :: String -> ClientM Bool  
     evaluate :: String -> ClientM Float  
       
     check :<|> evaluate = client api  
       
     baseUrl :: BaseUrl  
     baseUrl = BaseUrl Http "localhost" 3000 ""  
       
     data Action = Check | Evaluate deriving (Show, Read)  
       
     printResponse :: Show b => Either ServantError b -> IO ()  
     printResponse = either (putStrLn . ("Error: " ++) . show . SC.responseBody) print  
       
     performAction :: String -> Action -> IO ()  
     performAction expr action =  
      let manager = flip ClientEnv baseUrl <$> newManager defaultManagerSettings  
      in manager >>= \m -> case action of  
      Check -> printResponse =<< runClientM (check expr) m  
      Evaluate -> printResponse =<< runClientM (evaluate expr) m  
       
     main :: IO ()  
     main = do expr <- prompt "Expression: "  
      action <- prompt "Action (Check or Evaluate): "  
      maybe (print "Invalid action!") (performAction expr) $ readMay action  
      where prompt str = putStr str >> hFlush stdout >> getLine
   * Using Servant you can generate documentation and client side code from your API type. So I declared functions check and evaluate and then used pattern matching to assign the automatically derived implementations from the API type. ClientM is the monad in which client functions run.
   * API type doesn't contain any data about where the web server is hosted, so to run the derived querying functions you need to specify base URL. I defined a variable baseUrl that stores URI scheme, host, port, and base path.
   * Then I defined a data type that represents an action (Check or Evaluate) and derived instances of Show and Read for it.
   * runClientM :: ClientM a -> ClientEnv -> IO (Either ServantError a)
   * As you can see runClientM, the function used for running queries, returns IO (Either ServantError a). To show responses I wrote a function printResponse.
   * either :: (a -> c) -> (b -> c) -> Either a b -> c  
     SC.responseBody :: ServantError -> Data.ByteString.Lazy.Internal.ByteString
   * printResponse prints the response if no errors occurred and prints "Error: <error message>" if the server responded with an error.
   * data ClientEnv = ClientEnv { Servant.Common.Req.manager :: Manager  
      , Servant.Common.Req.baseUrl :: BaseUrl  
      } -- Defined in ‘Servant.Common.Req’  
     newManager :: ManagerSettings -> IO Manager  
     defaultManagerSettings :: ManagerSettings  
       
     flip ClientEnv baseUrl <$> newManager defaultManagerSettings :: IO ClientEnv
   * performAction takes an expression and an action. It queries the server to perform the right action and then prints the response using printResponse.
   * main prompts user for an expression and an action, parses the action using readMay and performs the action if it is valid, otherwise it prints "Invalid action!". I defined a local function prompt that takes a string, prints it, and reads a line from the user. hFlush causes any buffered items to get sent immediately to the operating system. We need to call hFlush because by default due to Haskell's use of lazy IO the standard output data gets sent to the OS only after we print a new line or the buffer is full.
2. Ruby
   1. Server
   * For the Ruby version of the server I used a popular framework for developing web application called Ruby on Rails. I generated a new rails project and a new controller. Rails uses MVC (model, view, controller) model. Models are used for working with data, views render data, and controllers have the logic of the application.
   * $ rails new ruby --api  
     $ rails generate controller Main
   * This application is very simple, so all I need is one controller, so I changed config/routes.rb to tell Rails to use MainController class for all requests.
   * Rails.application.routes.draw do  
      post '/:action(/:id)', controller: 'main' # route all requests to the main controller  
     end
   * Then I wrote MainController.
   * # class that handles HTTP requests  
     class MainController < ApplicationController  
      # /check  
      def check  
      # .nil? returns true if the object is nil  
      # parse the expression from the request and return boolean  
      # in JSON format that shows if the expression is valid or not  
      render json: !parse(JSON.parse(request.body.read)).nil?  
      end  
       
      # /evaluate  
      def evaluate  
      # check if the expression is valid  
      if (expr = parse(JSON.parse(request.body.read))).nil?  
      # if it's invalid respond with an error  
      render body: 'Invalid expression!', status: 400  
      else  
      # evaluate the expression and return the result in JSON  
      render json: eval\_expr(expr)  
      end  
      end  
       
      # everything below is private  
      private  
       
      # class that represents a simple mathematical expression  
      class Expr  
      # getters and setters for a binary operator and two operands  
      attr\_accessor :operator, :operand1, :operand2  
       
      # simple class constructor  
      def initialize(operator, operand1, operand2)  
      self.operator = operator  
      self.operand1 = operand1  
      self.operand2 = operand2  
      end  
      end  
       
      # function for evaluating expressions  
      def eval\_expr(expr)  
      # return nil if the given expression is nil  
      nil if expr.nil?  
       
      case expr  
      when Expr # when the expression is an instance of Expr  
      # evaluate the operands and apply the operator to the results  
      case expr.operator  
      when '+'  
      eval\_expr(expr.operand1) + eval\_expr(expr.operand2)  
      when '-'  
      eval\_expr(expr.operand1) - eval\_expr(expr.operand2)  
      when '\*'  
      eval\_expr(expr.operand1) \* eval\_expr(expr.operand2)  
      when '/'  
      eval\_expr(expr.operand1) / eval\_expr(expr.operand2)  
      end  
      else  
      # when the expression is not an instance of Expr it should be a number  
      # return the number  
      expr  
      end  
      end  
       
      # function for parsing expressions  
      def parse(str)  
      exprs = [] # array of expressions used as a stack  
      buffer = '' # buffer for parsing numbers  
       
      # loop through each character  
      str.each\_char do |d|  
      # we can apply operators only if there are at least two expressions in the stack  
      if exprs.length >= 2  
      # if the current character is an operator then take first two elements  
      # from the stack, construct a new expression, and put it in the stack  
      case d  
      when '+'  
      exprs.unshift Expr.new '+', exprs.shift, exprs.shift  
      when '-'  
      exprs.unshift Expr.new '-', exprs.shift, exprs.shift  
      when '\*'  
      exprs.unshift Expr.new '\*', exprs.shift, exprs.shift  
      when '/'  
      exprs.unshift Expr.new '/', exprs.shift, exprs.shift  
      when ' '  
      # if the buffer isn't empty then there is a number in it  
      unless buffer.empty?  
      begin # try  
      x = Float(buffer) # convert to float  
      exprs.unshift x # put in the stack  
      buffer = '' # empty the buffer  
      rescue(ArgumentError) # catch parsing exception  
      # the expression is invalid, break the loop  
      break  
      end  
      end  
      else  
      # put the character in the buffer  
      buffer << d  
      end  
      else # less than two elements in the stack  
      # only need to check if the character is ' '  
      case d  
      when ' '  
      # the same behavior in case of a space  
      unless buffer.empty?  
      begin  
      x = Float(buffer)  
      exprs.unshift x  
      buffer = ''  
      rescue(ArgumentError)  
      break  
      end  
      end  
      else  
      # put the character in the buffer  
      buffer << d  
      end  
      end  
      end  
       
      # if the buffer is empty and the expressions stack has only one element  
      # return the expression  
      exprs.shift if exprs.length == 1 && buffer.empty?  
       
      # expressions in reverse polish notation should have an operator at the end,  
      # so if the buffer isn't empty then the expression is invalid.  
      end  
     end
   1. Client
   * require 'excon' # library for HTTP  
     require 'json' # library for JSON  
       
     # function for querying /check  
     def post\_check(excon, body)  
      excon.request(  
      method: :post,  
      path: '/check',  
      headers: { 'Content-Type' => 'application/json' },  
      body: body  
      )  
     end  
       
     # function for querying /evaluate  
     def post\_evaluate(excon, body)  
      excon.request(  
      method: :post,  
      path: '/evaluate',  
      headers: { 'Content-Type' => 'application/json' },  
      body: body  
      )  
     end  
       
     # expression input  
     print 'Expression: '  
     expr = JSON.generate gets.chomp # .chomp removes carriage return characters (like \n)  
       
     # action input  
     print 'Action (Check or Evaluate): '  
     action = gets.chomp  
       
     # creating an instance of excon with base url http://localhost:3000  
     excon = Excon.new('http://localhost:3000')  
       
     case action # identifying the action  
     when 'Check'  
      res = post\_check(excon, expr)  
      print 'Error: ' if res.status != 200  
      puts res.body  
     when 'Evaluate'  
      res = post\_evaluate(excon, expr)  
      print 'Error: ' if res.status != 200  
      puts res.body  
     else  
      puts 'Invalid action!'  
     end

### Tests

1. Haskell

* 

1. Ruby
   1. Server
   * 
   1. Client
   * 

### Evaluation

1. Safety

* In this example I defined an API using types. This approach ensures that the implementation of the server corresponds to the API definition. For example, we can be absolutely sure that /check takes a string and returns a boolean. In the ruby version of the server we don't have that safety. We can accidentally return the result in a wrong format or return nothing at all. Also, with this approach you only use the parameters of the request that you specified in the API type. For example if all you need from a request is the body then you define the type with ReqBody and get the contents of the request body. If the request data that you can access is restricted by the type then there are less mistakes you can possibly make when you implement the API. When you are implementing your API using Servant you only think about the logic of your code, things like encoding and decoding data are done automatically through type classes. This also reduces the number of possible bugs.

1. Expressiveness

* Just by looking at the number of lines of code that I wrote we can see how Haskell is more expressive.

|  |  |
| --- | --- |
| * Language | * Lines of code (client + server) |
| * Haskell | * 108 |
| * Ruby | * 179 |

* Keep in mind that this is all the Haskell code, but I didn't include 380 lines of ruby code in different configuration files generated by the framework. The reason I didn't include them is that Ruby on Rails framework is an overkill for the problem that I was solving. The Haskell version of the code not only uses less amount of code, but it is also almost exactly matches the algorithms' descriptions in English in <sec:design_reverse_polish>. For example, let's take a look at the parse function.
* parse :: (Read a, Fractional a) => String -> Maybe (Expr a)  
  parse = flip parseAccum [] . words  
   where parseAccum :: (Read a, Num a) => [String] -> [Expr a] -> Maybe (Expr a)  
   parseAccum [] [x] = Just x  
   -- "if there is one element in the stack then return it"  
    
   parseAccum ("+":cs) (x1:x2:xs) = parseAccum cs $ x1 :+: x2 : xs  
   parseAccum ("-":cs) (x1:x2:xs) = parseAccum cs $ x1 :-: x2 : xs  
   parseAccum ("\*":cs) (x1:x2:xs) = parseAccum cs $ x1 :\*: x2 : xs  
   parseAccum ("/":cs) (x1:x2:xs) = parseAccum cs $ x1 :/: x2 : xs  
   -- "when it sees an operator in the input it takes two expressions  
   -- from the stack and constructs a new expression with the operator  
   -- it read as the operator and the two expressions as the operands  
   -- and puts the new expression in the stack"  
    
   parseAccum (str:cs) exprs = readMay str >>= parseAccum cs . (: exprs) . Number  
   -- the function treats the rest of the input as numbers  
    
   parseAccum \_ \_ = Nothing  
   -- the expression is invalid
* As I mentioned in the evaluation of safety the implementation of the API contains only the logic of the application. This feature not only makes the code safer, but also makes it more expressive. This shows that Haskell is very expressive.

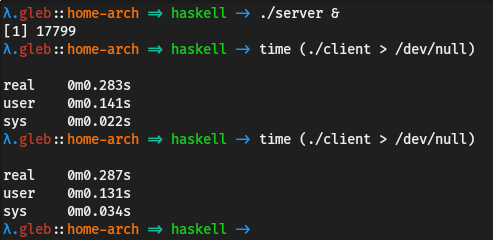
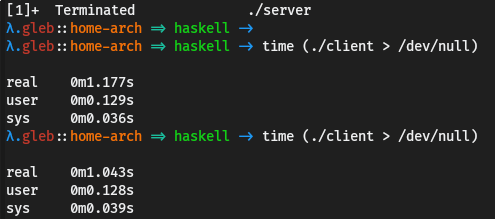
1. High level abstractions

* Defining an API in terms of types is a great example of Haskell's high level abstractions. In most languages even if you have a type system you can't do anything like this. First of all you can't define readable type-level grammar. Also even if you somehow manage to define types that can be used for constructing a type that represents an arbitrary REST API, the constructed type will be useless because the majority of languages don't support type-level functions. So you won't be able to type check the implementation of your API.
* The point I made about only describing logic in the implementation of the server is also relevant to this discussion. The way Servant encodes and decodes data to the right formats just by looking at the type definition of your API is also a great example of using abstractions in Haskell. But you can go even further and define an API type using type variables and then implement it once for an arbitrary type variable that is constraint by type classes that are required for encoding, decoding, etc. This can be useful if, for example, you need the same CRUD (create, read, update delete) API for several entities. All these features of the type system give Haskell support for all kinds of different abstractions.

1. Modularity

* The fact that you can use types to define APIs is good for modularity. You can define parts of your API and then compose them. Also you can use the approach I mentioned in the previous section.
* -- ...  
  type Create a = ReqBody '[JSON] a :> Post '[JSON] a  
    
  type Read a = Get '[JSON, PlainText] a  
    
  data User = -- ...  
    
  type UserAPI = Create User :<|> Read User  
    
  userServer :: Server UserAPI  
  userServer = -- ...  
    
  data Mail = -- ...  
    
  type MailsAPI = Create Mail :<|> Read Mail  
    
  mailServer :: Server MailAPI  
  mailServer = -- ...  
    
  type API = "users" :> UserAPI  
   :<|> "mails" :> MailAPI  
    
  server :: Server API  
  server = userServer :<|> mailServer
* As you can see defining APIs using types is good for modularity. Haskell itself is very modular. And its high level abstractions allow developers to use standard (more or less) tools when they develop libraries. So, for example, grammar for defining APIs can be embedded directly into the language. This means that you can write modular code and then easily reuse it. You can also extend the grammar by providing the required type class instances for your types and type combinators. This shows that Haskell has good modularity.

1. Performance

* Let's query the /evaluate endpoint of both of the servers 1000 times with the same expression 15 7 1 1 + - / 3 \* 2 1 1 + + - and see how much time it will take for each of the servers. To do this I changed the end of the client script. I added repeatAction function that takes an integer n and a monadic action, and runs the action n times.
* repeatAction :: Monad m => Integer -> m () -> m ()  
  repeatAction n a = foldr (const (>> a)) (return ()) [1..n]  
    
  performAction :: String -> Action -> IO ()  
  performAction expr action =  
   let manager = flip ClientEnv baseUrl <$> newManager defaultManagerSettings  
   in manager >>= \m -> case action of  
   Check -> printResponse =<< runClientM (check expr) m  
   Evaluate -> repeatAction 1000 $ printResponse =<< runClientM (evaluate expr) m  
    
  main :: IO ()  
  main = performAction "15 7 1 1 + - / 3 \* 2 1 1 + + -" Evaluate
* Here are the results of the benchmark that show that the Haskell version of the server worked more than 3 times faster than the Ruby version.
  1. Haskell
  + 
  1. Ruby
  + 

# Conclusion

## Safety

Haskell allows you to encode logic in types. The benefits of this approach is that at compile time your code is checked and it must follow the types, so if your logic is encoded in types then the compiler checks that your code follows your logic. I showed this with the web server example, there I encoded the API in types. Then when I implemented the API the compiler checked that the code implements the specified API.

Using Haskell you can avoid almost all run-time errors. For example, if you have a function that can fail then you encode that fail in types by wrapping the result in Maybe, Either a or another monad. This way when you actually use the function the compiler will force you to check that the function didn't fail, otherwise the code won't type check.

Immutability also makes a big role in making Haskell a very safe language. A lot of the errors in imperative languages are caused by state change. In most languages you can change a global variable in one function without realizing that this will break another function.

If you declare the type of your function or variable before the value then the number of possible ways in which you can write this variable or function is significantly reduced. There are few ways of writing the right code and many ways of writing incorrect code, in most languages the number of ways to write incorrect code is infinite. Haskell reduces the number of ways in which you can write your code, making it harder to write code that compiles, but doesn't work as it should.

data Bool = True | False  
  
not :: Bool -> Bool

There are four ways of defining the function not (2 possible inputs and 2 possible outputs for each of them). In languages like Ruby there are infinitely many ways of defining a function like this, just because the language doesn't have a type system and it uses impure functions. In languages like C# there are still infinitely many ways of defining not. The type system reduces the number of possible inputs and outputs, but you still can do any impure computations inside the function. You can choose to write some code that is not directly related to the function not and crash the application, for example by just dividing by zero. In Haskell to cause a problem this way you would need to put the division somewhere in the code and also force Haskell to evaluate the division, because Haskell won't evaluate the expression with division by zero as the result of the function doesn't depend on it. This is another example of how in Haskell it's harder to write incorrect code. Another example is maybe function.

maybe :: b -> (a -> b) -> Maybe a -> b

There are only two ways of defining this function (return the first argument or do the right thing). Because the function's type uses arbitrary type variables you can't just construct a new value of type b or do modify somehow the first two arguments. And sometimes type declaration uniquely defines what the function does. Below there are three functions that you can't implement incorrectly.

id :: a -> a  
const :: a -> b -> a  
flip :: (a -> b -> c) -> b -> a -> c

All these examples show how Haskell's type system and immutability make your code safer.

## Expressiveness

Haskell types not only make code safer, but they also make it more readable. Type signature significantly reduces the number of possible implementations of the function, so if you know the name and the type of a function then more often than not you can tell what the function does without looking at the implementation. This helps a lot when you are working in a team and you want to use a function written by another person. It also makes it easier to use libraries. By defining types using newtype and type you can make it even easier to understand what functions do without using any more data at run time.

As you saw in the examples, Haskell usually requires less code to solve a problem. The reason for this is the minimalist syntax and features like partial application, pattern matching, custom operators, etc. Also, in both cases the code was close to the description of the problem in English. This shows that Haskell is a very expressive language.

## High level abstractions

## Modularity

## Performance

# Links

<http://benchmarksgame.alioth.debian.org/u64q/haskell.html>

<https://wiki.haskell.org/Why_Haskell_matters>

<https://github.com/Gabriel439/post-rfc/blob/master/sotu.md>