# Purpose

The purpose of this project is to investigate how Haskell works and how its features benefit the developers. 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.

## 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 a big problem - it clutters up the language.

## High level abstractions

To write programs as fast as possible we want to describe our ideas without thinking about every single operation that 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.

## Modularity

We want to reuse as much code as possible. We also should be able to easily compose different parts of a program together.

## Performance

Depending on what we are developing we have different performance requirements, so we want programming languages to help us make efficient programs.

## Easy refactoring

In real world projects it's impossible to write a piece of code once and then never change it. This is why we want to be able to change existing code and to add new features without breaking anything.

Automated tests can help a lot with this problem too, but they have mentioned above problems.

# Design

Let's first take a look at how Haskell works and then I'll show how it's design is used to solve the identified above problems.

## 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.

### Arithmetic operations

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

### Logic

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

### Powers

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

### 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]

### 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.

### 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

### 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

### 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

### 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.

### 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

### 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)

### 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)..

(Reference: Learn Haskell Fast and Hard) ((I'll do all the references later))

## 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 () = ()

### 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!"

### 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 " ++) . (++ "!")

### 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

# Solution

Now I will solve several problems in Haskell as well as C and Ruby to compare the parameters of programming languages identified in analysis.

## Automatic differentiation using dual numbers

For differentiating mathematical functions we can use dual numbers. In maths we define dual numbers this way:

, where

If we have a function that is defined for real numbers and we pass a dual number into it then we'll get this

This property is used for automatic differentiation.

### Safety

### Expressiveness

### Abstractions

### Modularity

### Performance

### Refactoring

## Sort

In this example I will discuss a Haskell script that I wrote. It reads comma-separated numbers from a file, sorts them, and writes into another file using the same format.

{-# 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.

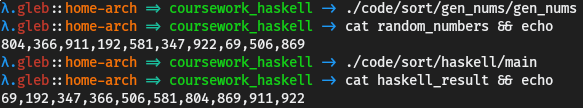
C.readFile "random\_numbers" :: IO C.ByteString  
  
maybe (print "Failed to parse!")  
 ( C.writeFile "haskell\_result"  
 . C.intercalate "," . fmap (C.pack . show) . sort . fmap fst  
 ) . traverse C.readInt . C.split ','  
 :: C.ByteString -> IO ()  
  
(>>=) :: Monad m => m a -> (a -> m b) -> m 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".

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

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

Let's test the code:



### 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.

### Expressiveness

As you can see we didn't need a lot of code to solve the problem. Let's describe the algorithm in English.

We want 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 "haskell\_result".

And 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" -- we want to 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

Let's compare this to an imperative solution of the problem in 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

Here, as you can see, 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.

### 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.

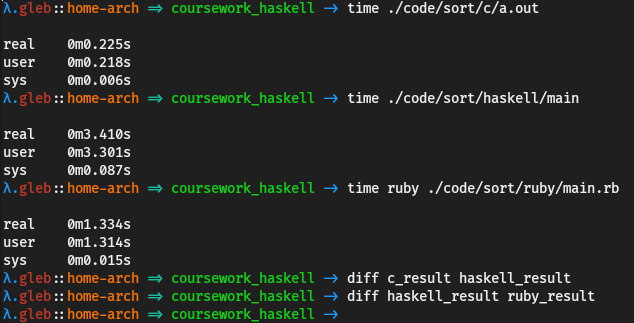
{-# LANGUAGE OverloadedStrings #-}  
  
import qualified Data.ByteString.Lazy.Char8 as C  
import Data.List (sort)  
  
parse :: C.ByteString -> Maybe [Int]  
parse = fmap (fmap fst) . traverse C.readInt . C.split ','  
  
main :: IO ()  
main = C.readFile "random\_numbers"  
 >>= maybe (print "Failed 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.

### Performance

I solved the problem in C, so that I can compare the performance of C, Haskell, and Ruby.

#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;  
}



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.

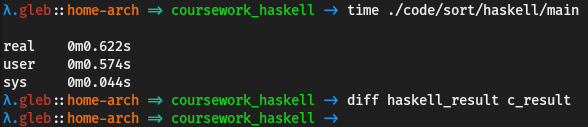
1. Introsort

* 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.](https://github.com/gperftools/gperftools)