# A-level Computer Science Project

# Type safe functional programming in Haskell

# Gleb Dianov

# 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.

## Concurrency

We want to be able to write safe concurrent applications.

# 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)..

((This will be a proper diagram))

(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.

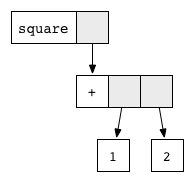
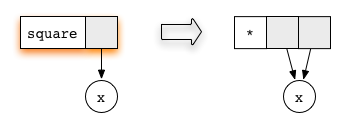
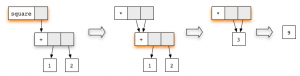
headMay :: [a] -> Maybe a  
headMay (x:xs) = Just x  
headMay [] = Nothing  
  
headOfHead :: [[a]] -> Maybe a  
headOfHead l = do h <- headMay l  
 headMay h

## Lazy evaluation

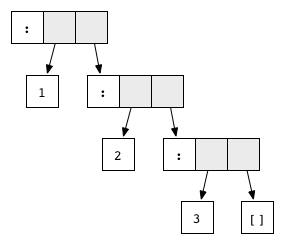
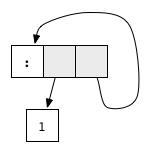
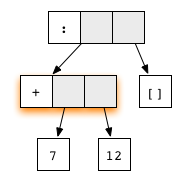
Haskell has a very interesting evaluation strategy. It doesn't execute expressions until it needs the result. It can make our code simpler and more modular, but it can also be confusing when it comes to estimating performance and memory usage. For example this simple expression that sums all numbers from 1 to 108 foldl 0 [1..10^8] requires gigabytes of memory to evaluate. But if we import the strict version of this function foldl' from the Data.List module and use it instead, everything's OK.

### How lazy evaluation in Haskell works?

1. Graph reduction

* Haskell programs are executed by evaluating expressions. The primary idea is function application. Here is a simple function:
* square x = x\*x
* Let's see how the following expression gets evaluated:
* square (1+2)  
  => (1+2)\*(1+2) -- replacing the left hand side  
  => 3\*(1+2)  
  => 3\*3  
  => 9
* We calculated (1+2) twice, to avoid that we use graph reduction method. In this graph every block is a function application. Our situation can be represented by the following graph:
* 
* This representation is similar to the way the compiler actually represents expressions with pointers. When a programmer defines a function they define a reduction rule, then when the function is applied the graph gets reduced until it becomes a basic expression. Any expression can be represented using graphs.
* Our function corresponds to this rule:
* 
* x is a placeholder for a subgraph. And when arguments get duplicated they point to the same subgraph, hence identical graphs don't get reduced multiple times.
* Any subgraph that follows the rules is called a reducible expression or redex. In our case with have two redexes: function square and addition +. If we start with square then we'll get this:
* 
* At every step the highlighted rectangle gets updated.

1. Normal form

* If the graph is not a redex then it means that we already reduced everything and got the result that we wanted. In the last example the normal form was a number, but constructors of algebraic data types like Just, Nothing, or lists constructors : and [] are not reducible. Even though they are functions they can't be reduced, that's because they were defined using data and don't have a right-hand side. For example, graph:
* 
* By definition a normal graph needs to be finite and it shouldn't have cycles. Infinite recursion is not normal.
* ones = 1 : ones
* Corresponds to the following cyclic graph.
* 
* It's not a redex and also not in the normal form - the tail of the list points to the list itself, making an infinite recursion.
* In Haskell expressions usually don't get to the normal form. Quite often we stop when we get to the weak head normal form (WHNF). A graph is in WHNF if it's top node is a constructor. Like expression (7+12):[] or graph
* 
* is in WHNF, its top node is a list constructor ((:)). And it's not the normal form because the first argument is a redex.
* The list ones is also in WHNF, its top node is a constructor. In Haskell we can create and use infinite lists.

### Execution order, lazy evaluation

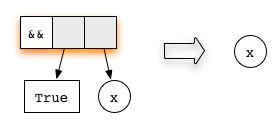
Often expressions have multiple redexes. Does the order at which we reduce them matter?

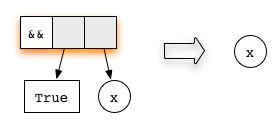
Most languages use the strategy that reduces arguments to the normal form before reducing the function, this is called eager evaluation. However, most Haskell compilers use a different evaluation order called lazy. It first reduces the top function application. That may require calculating some of the arguments, but only as many as it needs. Let's take a look at this expression with pattern matching. The arguments will get evaluated from left to right until the top node contains a constructor. If pattern matching isn't used then the arguments don't get evaluated. If you pattern match a constructor then the argument gets reduced to WHNF.

For example:

(&&) :: Bool -> Bool -> Bool  
True && x = x  
False && x = False

This defines two reduction rules:





Now let's take a look at this expression:

('H' == 'i') && ('a' == 'm')

Both of the arguments are redexes. Because of pattern matching the first argument will get evaluated. Then the graph will get reduced without evaluating the second argument.

### Performance

It's mathematically proved that lazy evaluation requires fewer or the same number of reductions (calculations) as eager evaluation. Also, in some cases, it can compute expressions with errors without crashing, such as

a = 1  
b = 2  
(a == b) && (1 == (b/0))

The second argument of (&&) will never get evaluated, hence the second argument of the second (==) will never get evaluated, thus we will never divide by zero and get an exception.

However, the memory usage is a tricky problem. Sometimes an expression reduced to normal form can use more memory than a redex, and vice versa. Let's take a look at examples of both cases.

enumFromTo 1 1000

This expression generates a list with numbers from 1 to 1000. The list itself takes much more space than the expression.

Here is another example:

((((0 + 1) + 2) + 3) + 4)

The graph that represents this expression takes more space than the normal form of the expression - 10.

However Haskell allows you to force reduction using the seq combinator.

seq :: a -> b -> b

If you look at the type signature you may think that it's exactly the same as the const function (with arguments in a different order), however they are not the same. seq reduces the first argument to the WHNF and then returns the second argument. const doesn't do anything with the first argument. It's important to remember that seq doesn't reduce the first argument to the normal form. For example, if we are reading a list of lines l from a file, we can't just use seq l to force Haskell to finish reading the list. This would just force it to read the first line, because that's enough to know the constructor. To force Haskell to finish reading the file we need to use seq (length l). The only way to reduce length l to the weak head normal form is to find the length, hence to read the entire file. But in other cases this might not work, for example length $ (+7) <$> [1..10] will find the length without adding any numbers.

Here is a standard use case of seq that every Haskell programmer should know - strict left fold. Here is how foldl is defined in Prelude:

foldl :: (a -> b -> a) -> a -> [b] -> a  
foldl f a [] = a  
foldl f a (x:xs) = foldl f (f a x) xs

Say we want to sum all integers from 1 to 100 ([1..100]). For that we would use the expression foldl (+) 0 [1..100]. Here is how the evaluation process would look in that case:

foldl (+) 0 [1..100]  
=> foldl (+) 0 (1:[2..100])  
=> foldl (+) (0 + 1) [2..100]  
=> foldl (+) (0 + 1) (2:[3..100])  
=> foldl (+) ((0 + 1) + 2) [3..100]  
=> foldl (+) ((0 + 1) + 2) (3:[4..100])  
=> foldl (+) (((0 + 1) + 2) + 3) [4..100]  
...

As you can see the second argument accumulates a massive expression without reducing it, this causes high memory usage. To deal with this problem we need to keep the accumulator in WHNF. Here is how we can do this:

foldl' :: (a -> b -> a) -> a -> [b] -> a  
foldl' f a [] = a  
foldl' f a (x:xs) = seq a' $ foldl' f a' xs  
where a' = f a x

This function is defined in the module Data.List. Now evaluation will look like this:

foldl' (+) 0 [1..100]  
=> foldl' (+) 0 (1:[2..100])  
=> foldl' (+) 1 [2..100]  
=> foldl' (+) 1 (2:[3..100])  
=> foldl' (+) 3 [3..100]  
=> foldl' (+) 3 (3:[4..100])  
=> foldl' (+) 6 [4..100]  
...

During evaluation the expression has constant memory usage.

In a language with eager evaluation, like Python, it's impossible to write this function. In such language the list gets reduced to normal form before summing. This uses the same amount of memory as the inefficient version of foldl.

Let's take a look at how we can define [n..m].

enumFromTo n m = if n < m then n : enumFromTo (n+1) m  
 else []

So the reduction of [1..100] to WHNF actually looks like this:

[1..100]  
=> 1:[(1+1)..100]

So the new argument is not 2, it's (1+1). This shows us that it's very hard to predict how exactly expressions are evaluated. The actual definition of enumFromTo is different from the code above.

(Reference: How lazy evaluation works in Haskell) ((I'll do the references later))

# Solution

## Expressiveness

## Safety

## High level abstractions

## Modularity

## Performance

## Easy refactoring

## [Concurrency](https://wiki.haskell.org/Why_Haskell_matters)