

Vadim Prokopev

LAMBDA CALCULUS AND FUNCTIONAL PROGRAMMING

Using Haskell

Bachelor's thesis
Information Technology

2017



South-Eastern Finland
University of Applied Sciences

Author	Degree	Time
Vadim Prokopev	Bachelor of Information Technology	December 2017
Title		67 pages
Lambda Calculus and functional programming Using Haskell		
Commissioned by		
Jari Kortelainen		
Supervisor		
Jari Kortelainen		
Abstract		
<p>This thesis explores the world of Lambda Calculus, functional programming, which is based on Lambda Calculus, and Haskell as an example of a functional programming language. Lambda Calculus is a formal system for denoting something computable. It has played an important part in the development of the mathematical logic and was first introduced by Alonzo Church.</p> <p>Functional programming is one of the paradigms of programming. Other existing paradigms include imperative and object-oriented ones. Functional programming influenced languages from other paradigms, languages like C++ and Java have functional features added in the latest revisions.</p> <p>The aim of the thesis was to learn in which tasks functional programming is favorable. When Haskell would be preferable to other possible languages, or when some specific task should be performed using functional features and thus a language with the functional features is preferable.</p> <p>Several small programs were written in Haskell, Java and Python and then their performance got measured. The functional approach was used in Java and Python where it was considered more concise. Also, some subjective qualities like difficulty of writing or readability of code of programs were evaluated.</p> <p>The conclusion of research conducted showed that functional programming provides considerable amount of utility. However, using the purely functional approach, namely Haskell, is too restricting. For maximal efficiency mixed languages that were developed with the functional features as one of the core components like Scala should be used if functional features are required for the task at hand.</p>		
Keywords		
Lambda Calculus, functional programming, Haskell		

CONTENTS

1	INTRODUCTION	4
2	LAMBDA CALCULUS	5
2.1	Basic syntax.....	5
2.2	Introduction to a recursion	8
2.3	Church Encoding	9
2.4	Typed Lambda Calculus	10
2.5	Function signatures	11
3	HASKELL	12
3.1	Syntax.....	13
3.2	Data structures	17
3.3	Types and functions.....	20
3.4	Additional functional operations	28
3.5	Monads.....	32
4	CONFIGURING THE ENVIRONMENTS	42
4.1	Configuring Haskell.....	43
4.2	Configuring Java.....	44
4.3	Configuring Python	45
5	APPLICATION	46
5.1	Summing test.....	47
5.2	Recursion test.....	49
5.3	Derivatives list.....	51
5.4	Dijkstra test	55
5.5	Linear equations test	58
6	CONCLUSIONS	62
	REFERENCES	64

1 INTRODUCTION

Functional programming started its history with Lisp in 1950s which was followed by other languages like Algol 60 or PAL (Turner 2012). Since then functional programming walked a long path to become what it represents now. The basics of functional programming is Lambda Calculus which will be discussed in the section 2.

Haskell was the first purely functional language. It is specified and developed by special committee starting from 1987. The first release of Haskell was in 1990 and had several major version and revisions since then. The next major revision of Haskell is planned to be released in 2020 (Riedel 2016). Haskell will be discussed in the section 3.

The general idea of the research behind the thesis is to explore functional programming on the example of Haskell. It will be explored in the perspective of Lambda Calculus and other languages. Java and Python will be used for the comparison with Haskell. Java is the best known as an objective-oriented programming (OOP) language, but the revision to Java 8 implemented functional features (Oracle 2014). Python is a script language that includes functional and OOP features.

The aim of the thesis is to understand if functional programming is viable, i.e. is there applications when functional programming is superior to the imperative approach. This will be done by writing several small programs for solving quite simple various tasks in the field of mathematics. The points of the comparison will be the following:

- Readability and understandability of a code
- Execution time
- The maximum amount of a space required during an execution
- Difficulty of the implementation

Preparations required to perform the comparison will be discussed in the section 4 and comparison itself will be performed in the section 5.

2 LAMBDA CALCULUS

This section is mainly based on book by Michaelson (2011).

Lambda Calculus was invented by Alonzo Church when he tried to find the foundation of mathematics and how functions calculate (Jung 2004). His theory says that every function is a black box which consume some inputs and produce a desired output. Lambda Calculus as it was described by Church, is nowadays known as Untyped Lambda Calculus as there is no distinctive types of data like String or Integer, only functions.

Church showed that it is possible to compute anything using his system. It means that Lambda Calculus is a Turing-complete language, i.e. anything that may be computed by a Turing machine also may be calculated using Lambda Calculus (Rowland 2017).

2.1 Basic syntax

Lambda Calculus consists of lambda terms. Table 1 shows all possible lambda terms. There are not much of them as Lambda Calculus is using the very compact syntax.

Table 1. Syntax of possible lambda terms in Lambda Calculus

v	Variable	Character or string that represents some value
$(\lambda v. M)$	Function	Definition of function, the variable that is followed by λ is bound and M is a valid lambda term
$M N$	Application	Application of N to M where both are valid lambda terms

Let's compare classical mathematical, Java and Lambda Calculus ways of defining functions in an example of an addition seen in Table 2.

Table 2. Ways of defining functions using different notations

Lambda Calculus	$\lambda x. \lambda y. x + y$
Mathematical	$f(x, y) = x + y$
Java	<code>int f(int x, int y) {return x + y; }</code>

In a function binder is some formal symbol that bind variables from the outside world to the body of a function. In mathematics and Java, it is the name of a function which is f in the current case. There are no function names in Lambda Calculus and variables are getting bind by λ symbol which is always the same. Non-bound variables are called free.

Java like the most of other programming languages define types of inputs and of the output when Lambda Calculus is untyped and everything in it is a function. Absolutely everything is a function including numbers and thus addition is a complex function. But to make everything less abstract and more understandable $+$ will be used as usual.

For shorter notation additional λ after first one is often omitted as follows:

$$\lambda x. \lambda y. x y = \lambda x. y. x y \quad (1)$$

Arguments for the functions are written after them and can be any valid lambda terms. Variables are replaced from left to right and in the following expression M be used as x and N will be used as y :

$$(\lambda x. y. x + y) M N \quad (2)$$

Example. Various lambda terms

$$\lambda x. y. x y z \quad (3)$$

$$\lambda v. x * v \quad (4)$$

$$\lambda x. y. x x y \quad (5)$$

These terms are valid, but they are not that useful. They just follow syntax we defined. Some basic functions should be presented to continue working with them. The most common ones are identity, application, double application and constant functions seen in Equations 6-9.

$$id = \lambda x. x \quad (6)$$

$$apply = \lambda f. x. f x \quad (7)$$

$$twice = \lambda f. x. f (f x) \quad (8)$$

$$const = \lambda x. y. x \quad (9)$$

The identity function returns an input unchanged. The application function takes a function and an argument and apply it to the function. The double application is the same, but runs function again with the result of the first application as an input. The constant function takes two inputs but always will return the first one. Formally they are defined in the following way:

$$id\ M = M \quad (10)$$

$$apply\ M\ N = M\ N \quad (11)$$

$$twice\ M\ N = M\ (M\ N) \quad (12)$$

$$const\ M\ N = M \quad (13)$$

Lambda Calculus is left associated, it means that expressions like $M\ N\ O$ are evaluated like $(M\ N)\ O$, that's why $twice \neq M\ M\ N = (M\ M)\ N$. If a different order of an application is required, then parentheses should be added.

To evaluate functions the beta-reduction should be used, as it is the only way to evaluate functions in Lambda Calculus. Another operation defined upon lambda terms is the alpha-conversion.

The formal definition of the beta-reduction is the following:

$$(\lambda x. M)\ N = M[N/x] \quad (14)$$

$M[N/x]$ means replacing all bound occurrences of x with N in expression M . To denote the beta-reduction operation \rightarrow_β will be used. The beta-reduction is applied in two steps. First, the leftmost λ with the respective variable are removed. Second, all the occurrences of that variable are replaced with the input given.

Example. Various examples of the beta-reduction

$$id\ apply = (\lambda x. x)\ apply \rightarrow_\beta apply \quad (15)$$

$$apply\ id = (\lambda f. \lambda x. f\ x)\ id \rightarrow_\beta (\lambda x. id\ x) \quad (16)$$

$$const\ M\ N = (\lambda x. y. x)\ M\ N \rightarrow_\beta (\lambda y. M)\ N \rightarrow_\beta M \quad (17)$$

There is a possible situation that an expression contains several functions with the same name for bound variables, and the name collision can happen. To solve that issue the alpha-conversion can help with the following formal definition:

$$(\lambda x. x) = (\lambda y. y)\ \{y/x\} \quad (18)$$

$\{y/x\}$ means renaming all bound occurrences of y with x .

There are two ways of applying the beta-reduction: normal order reduction from left to right or applicative order from the innermost function and it will not change output. The Church-Rosser theorem says that every expression that can be reduced has only one normal form, i.e. the order of the beta-reduction doesn't matter for the final answer (Jung 2004). However, this doesn't mean that reductions may be done in any order, only the ones with priority may be done.

2.2 Introduction to a recursion

The recursion has the utmost importance in Lambda Calculus, and to learn how to do it, it is worth first looking at several special lambda terms. Let's define the simple function of the self-application seen in Equation 19. This function takes any function given and apply it to itself.

$$sa = \lambda x. x x \quad (19)$$

Now, let's try to plug some functions to it.

Example. The self-application with various functions

$$sa\ id = (\lambda x. x x)\ id \rightarrow_{\beta} id\ id = (\lambda x. x)\ id \rightarrow_{\beta} id \quad (20)$$

$$sa\ twice = (\lambda x. x x)\ twice \rightarrow_{\beta} twice\ twice = \quad (21)$$

$$(\lambda f. \lambda x. f (f x))\ twice \rightarrow_{\beta} (\lambda x. twice (twice x))$$

$$sa\ const = (\lambda x. x x)\ const \rightarrow_{\beta} const\ const = \quad (22)$$

$$(\lambda x. y. x)\ const \rightarrow_{\beta} \lambda y. const$$

But what if we self-apply the self-application? It will give a very interesting but practically useless term that is called Ω term and can be seen in Equation 23. It represents an infinite cycle.

$$\Omega = sa\ sa = (\lambda x. x x)\ sa \rightarrow_{\beta} sa\ sa \quad (23)$$

This term will never reduce, i.e. attempts to reduce it will return the same term. This means that it doesn't have a normal form. But, it is possible to have an even worse case, a lambda term can explode infinitely and become only bigger when it's being reduced. The simplest example is presented in Equation 24.

Example. The simplest expanding by reduction lambda term

$$(\lambda w. w w w)(\lambda w. w w w) \rightarrow_{\beta} (\lambda w. w w w)(\lambda w. w w w)(\lambda w. w w w) \quad (24)$$

These two terms cannot be used directly to make the recursion, but give an intuition on problems that can occur in case of the wrong approach.

The recursion is a process of calling function from within itself. However, in Lambda Calculus functions cannot have access to the unbound version of itself, i.e. there cannot be a function definition like $g = \dots g \dots$, but it is possible to have access to itself using fixed-point combinators (Hudak 2008). There are infinitely many of them, but the most known and most often used is the Y-combinator of Equation 25.

$$Y = \lambda t. (\lambda x. t(x x))(\lambda x. t(x x)) \quad (25)$$

Let's try to plug some input and see what will happen in Equation 26.

$$Y t = (\lambda t. (\lambda x. t(x x))(\lambda x. t(x x))) t \rightarrow_{\beta} \quad (26)$$

$$(\lambda x. t(x x))(\lambda x. t(x x)) \rightarrow_{\beta}$$

$$t (\lambda x. t(x x))(\lambda x. t(x x)) =$$

$$t (Y t) = t(t(Y t)) = \dots$$

It's easy to observe that our expression exploded like in Equation 24, and therefore, it may represent an infinite cycle. To prevent it from exploding it is possible to use conditional statements. Yet, there is nothing in Lambda Calculus but functions. How there can be conditionals? The answer is Church Encoding.

2.3 Church Encoding

Church invented the rigorous system of how give meaning to functions (McCarthy 2012). There are all basic structures we are used to see in the modern programming languages: integers, lists, Booleans. Even further it captures not just plain values but whole structures like conditionals. This system is too thick to fit inside thesis like this and will be discussed in short to give a taste of it.

What are Boolean values in general? It is choice between a and b , so it makes sense to define the Boolean value in Lambda Calculus as a function that takes

two values and returns only one of them, the encoding for corresponding values may be seen in Equation 27 and 28.

$$True = \lambda a. \lambda b. a \quad (27)$$

$$False = \lambda a. \lambda b. b \quad (28)$$

The basic Boolean operations: *and* and *or* are defined in the following way:

$$and = \lambda p. \lambda q. p \ q \ p \quad (29)$$

$$or = \lambda p. \lambda q. p \ p \ q \quad (30)$$

They may look too strange and obscure but after some deductions they will start make sense. *True* returns the first argument, *False* returns the second argument. In case of the *and* operation if the first argument is *True* then it would return the second argument. If the second argument is also *True* then the final answer is *True*, else it's *False* following the classic Boolean logic. If the first argument is *False* then it will return itself and it's of course *False*, again following Boolean logic. Same procedure may be done with *or*.

Following same ideas numbers, strings and lists may be defined. Though, this is an interesting way to define data, it has not much practical usefulness. Now, we will move further to Typed Lambda Calculus which has not only functions but all other usual things like numbers.

2.4 Typed Lambda Calculus

Originally Alonzo Church depicted Lambda Calculus as a system where only functions exist, which we already named Untyped Lambda Calculus. The real meaning of function comes from the encoding, but it is more reasonable to apply ideas of Lambda Calculus to usual data types that can be found in programming: integers, float point numbers or strings to get Typed Lambda Calculus. The basic principle of the beta-reduction to get an answer is still applied but now it is possible to get a meaningful answer without the encoding. The simplest data type is an integer and the simplest not unary operation is an addition. So, an addition of an integer number and 1 is one of the simplest typed operation that is called the successor, usually shortened to *succ* and has the following definition:

$$succ = \lambda x. x + 1 \quad (31)$$

Example. The beta-reduction of the successor function

$$\text{succ } 6 = (\lambda x. x + 1) 6 \rightarrow_{\beta} 6 + 1 \quad (32)$$

The input 6 replaces the variable x and λx is removed just like in Untyped Lambda Calculus. Now, let's try to define the function that takes two arguments but apply only one and observe results.

Example. A partial application of the beta-reduction

$$(\lambda x. y. x + y) 3 5 \rightarrow_{\beta} (\lambda y. 3 + y) 5 \quad (33)$$

Using the beta-reduction x was substituted by the input 3 and the new function was created which now sums 3 and the input. This method of taking just one argument is the golden nugget of Haskell called currying and will be discussed further in the appropriate section. It is possible to give a name to this new function and use it as usually.

Example. Creating a function from a partial application

$$\text{addThree} = (\lambda x. y. x + y) 3 \rightarrow_{\beta} (\lambda y. 3 + y) \quad (34)$$

Now let's use the double application with the successor function and the integer 2 as inputs. The resulting equations with the applied beta-reductions may be seen in Equation 35. Note, x in the twice function was alpha-converted to y .

Example. The application of *twice* and *succ* functions with a number

$$\begin{aligned} \text{twice succ } 2 &= (\lambda f. \lambda y. f (f y)) (\lambda x. x + 1) 2 \rightarrow_{\beta} & (35) \\ & (\lambda y. (\lambda x. x + 1) ((\lambda x. x + 1) y)) 2 \rightarrow_{\beta} \\ & (\lambda x. x + 1)((\lambda x. x + 1) 2) \rightarrow_{\beta} ((\lambda x. x + 1) 2) + 1 \rightarrow_{\beta} 2 + 1 + 1 \end{aligned}$$

2.5 Function signatures

Adding types reduces an abstraction and implies that all functions have specific signatures for what types it takes. Let's define types for some functions we've been using so far in Equations 36-40. Arrow \rightarrow is just a separator.

$$id :: a \rightarrow a \quad (36)$$

$$const :: a \rightarrow b \rightarrow a \quad (37)$$

$$apply :: (a \rightarrow b) \rightarrow a \rightarrow b \quad (38)$$

$$twice :: (a \rightarrow a) \rightarrow a \rightarrow a \quad (39)$$

$$succ :: Integer \rightarrow Integer \quad (40)$$

Terms in parentheses denote functions that takes corresponding input and output values. If type is not set explicitly line in *succ* then there is no restriction to specific types, but there must be correlation between types. Different letters don't mean that types will be different, but the same letters mean that types must be the same. It should be noted that there is no distinction between inputs and the return value.

The identity function should return an object of the same type as it was given, because it is the same object without any changes. The constant function takes two arguments but always returns the first one, so type of second argument can be any while the return type is the same to the type of first argument as it returns it unchanged. The application function takes a function that should be able to use the second argument given, so both should be the same type and return type of the given function and the application function should be the same. The double application function should get a function which takes an argument and the return variable of the same type because the function should be able to run again using a result of the first run as an argument. Our successor function can operate only on numeric value and has integer signature. (Reddy 2009.)

We've discussed partial application and it is possible to divide expression at any arrow that is not inside parentheses to get the signature of the partially applied function, e.g. *apply* giving only one of two arguments will give the following:

$$apply\ f :: a \rightarrow b \quad (41)$$

3 HASKELL

This section is mainly based on the book by Lipovača (2011).

Lambda Calculus is just a basis for functional programming which in its own manner implement its logic and may differ between languages. Functional

programming inherits the main idea of Lambda Calculus of being all about functions and their compositions. Nowadays, a wide range of modern languages support functional operations like Java or Python, but for this thesis the most functional language was chosen - Haskell to emphasize the functional approach in the process of writing programs. Haskell is the general-purposed, strongly-typed, lazy-evaluated, purely functional language. It was named after logician Haskell Curry.

3.1 Syntax

The syntax of real Haskell looks alike our convenience notation with the named functions in Lambda Calculus, e.g. the left part of Equation 32 is already the valid Haskell syntax, but to be specific let define a simple function and discuss noticeable and not that obvious points of syntax by reimplementing the identity function as in Equation 6 in the following way:

$$id\ x = x \tag{42}$$

The name of a function must start with a lowercase letter followed by the arguments which is separated from an implementation part by the equal sign. There is no the *return* keyword (though, it will appear later) to return a value like in in Java or Python because in Haskell every function must return some value. By default, Haskell functions are prefix functions just like in Lambda Calculus. Let's redefine all the other basic functions we used in Lambda Calculus part in the following way:

$$apply\ f\ x = f\ x \tag{43}$$

$$twice\ f\ x = f\ (f\ x) \tag{44}$$

$$const\ x\ y = x \tag{45}$$

The one simplification that is possible to make is that *const* function names the second variable and doesn't use it. Haskell provides a way of ignoring variables by substituting it with the underscore as follows:

$$const\ x\ _ = x \tag{46}$$

Conditionals, which already were mentioned in Lambda Calculus part, are a little different in Haskell from other languages. It is usually required to write what happens if the predicate is true and to omit the so-called *else* part. In Haskell it doesn't hold, as every function should return something. Therefore, the pattern

becomes ‘if – condition – then – this – else – that’. The keyword *then* to separate the predicate from the first execution path is added as there are no parentheses or braces. The general syntax for conditional statements is presented in Equation 47. Terms in angle brackets denote placeholders for values.

$$if \langle predicate \rangle then \langle expression \rangle else \langle expression \rangle \quad (47)$$

Example. The application of an if statement in Haskell

$$if \ x < 100 \ then \ 2 * x \ else \ x \quad (48)$$

The important feature of Haskell’s if statements is that they are valid expressions, i.e. they may be written anywhere where expressions like $x + y$ may be written, which means that an if expression can contain another if expressions inside itself. But, what if we need to choose between many clauses like switch in other languages, which is preferred to the nested if statements? Haskell has its own case syntax that looks like the switch syntax in other languages, but it is not used that often. Instead, a syntactic sugar of multiple declarations is used that will compile to the case syntax. The multi-declaration syntax is helpful to declare corner cases. The multi-declaration has the following syntax:

$$\begin{aligned} &\langle function \ name \rangle \langle arguments \rangle = \langle expression \rangle \quad (49) \\ &[\langle function \ name \rangle \langle arguments \rangle = \langle expression \rangle] \dots \end{aligned}$$

There can be any number of function declarations as needed which is denoted by ... after brackets.

Example. The multi-declaration of the power function

$$\begin{aligned} power _ 0 &= 1 \quad (50) \\ power \ 0 _ &= 0 \\ power \ x \ y &= x * power \ x \ (y - 1) \end{aligned}$$

The underscore symbol as an argument means that we don’t use the value of this variable for choosing the path of the execution. The declaration with just named variables without specific values will catch all the remaining cases. The power function of any number to the integer power is defined as the multiplication of the base by itself power times. In programming it is usually achieved using cycles.

But, Haskell don't have cycles and a recursion should be used instead. To solve some task recursively the base case should be defined and then call that will converge to the base case. The base case is the power equals to 0 and then $x^0 = 1$. Otherwise, $x^y = x * x^{y-1}$ and we can multiply x by the x to the power minus one. The corner case of 0 as $0^y = 0$, except 0^0 case, which is not covered.

But what if we need to check several predicates? Then the guard syntax is exactly what we need. It is the system that holds predicates and test them one after another. The guard syntax is presented in Equation 51. Vertical lines should be aligned to each other, otherwise the compiler will not succeed in understanding it properly.

$$\begin{aligned} &<function\ name>\ <arguments> && (51) \\ &|\ <predicate>\ =\ <exression> \\ &[|\ <predicate>\ =\ <exression>] \dots \end{aligned}$$

As with the multi-declaration there can be as many guard statements as needed.

Example. The power function with the guard syntax

$$\begin{aligned} &power\ x\ y && (52) \\ &|\ y == 0 = 1 \\ &|\ x == 0 = 0 \\ &|\ otherwise = x * power\ x\ (y - 1) \end{aligned}$$

Usually, the last predicate is replaced by *otherwise* keyword. This keyword is just another name for *True* which is the simplest predicate. It will catch all cases when previous clauses didn't work out. Running a function that declared using the guard syntax and none of predicates returned *True* will cause an error.

The main thing in programming is variables. Variables in Haskell are immutable, if you set them you cannot modify only use it. This may sound limiting but tasks that usually are solved by Haskell are being more like described and not followed step by step. To introduce local variables *where* and *let* constructions are used. To create the *where* statement, keyword *where* is added in the end of a function and all required variables or even functions may be defined there as in Equation

53. It is often used to make complex functions less burdensome. All expressions should be aligned to each other.

$$\begin{aligned} &<function\ name> && (53) \\ &\quad where\ <expression> \\ &\quad\quad [<expression>] \dots \end{aligned}$$

There can be any number of expressions. Also, it is possible to write the *where* statement for the function declared inside another *where* statement.

Example. The function with the *where* statement

$$\begin{aligned} &cubeSurface\ side = 6 * squareSurface && (54) \\ &\quad where\ squareSurface = side * side \end{aligned}$$

The *let* construction has similar capabilities. But, with one advantage, it is a valid expression, i.e. it may be written anywhere where any expressions like $x + y$ and if statements can be placed. *let* can be written both line by line and inline and has two following syntaxes:

$$let\ <expression>\ [;\ <expression>] \dots\ in\ <expression> \quad (55)$$

$$let\ <expression> \quad (56)$$

$$[\ <expression>] \dots$$

$$in\ <expression>$$

In the inline syntax several expressions are separated by a semicolon, in the several line declaration same expressions should be aligned just like in the guard syntax.

Example. Previous example using *let*

$$\begin{aligned} &cubeSurface\ side = && (57) \\ &\quad let\ sideSurface = side * side \\ &\quad in\ 6 * sideSurface \end{aligned}$$

An important feature derived from Lambda Calculus is lambda or anonymous functions shown in Equation 58. They are helpful if some small function which will be used only once in any of three additional functional operations that will be defined in the appropriate section.

$$\backslash \langle arguments \rangle \rightarrow \langle function\ body \rangle \quad (58)$$

Backslash is used as it resembles λ to denote start of the anonymous function.

Example. The function that takes an argument and multiplies it by itself

$$\backslash x \rightarrow x * x \quad (59)$$

3.2 Data structures

There are two major data structures used in Haskell: lists and tuples. Lists are infinitely expandable but can contain only one type of objects inside. Lists are the main data structure in Haskell. Strings are also lists, and all list methods may be applied to them. Lists are declared by putting elements inside brackets as in Equations 60 and 61.

$$['a', 'b', 'c'] \quad (60)$$

$$[[1,2,3], [4], [8,9]] \quad (61)$$

But this is just a syntactic sugar for the way lists are defined. Lists are defined recursively as either an empty list or an element tailed by a list. Informally it may be shown as follows:

$$list = empty\ or\ element : list \quad (62)$$

The infix function `:` is a type constructor, but will be discussed in the next section. What is important, is that it is used to prepend elements to the existing list and it is right associative, so it is possible to chain them as follows:

$$1 : 2 : 3 : [] \quad (63)$$

Because of this it is possible to unroll lists into first values using the following syntax:

$$x : y : xs = a_0 : a_1 : [a_2 .. a_n] \quad (64)$$

Trying to unroll a list to more variables than there are present will result in an error. The main operations possible to do with lists are depicted in Table 3. There are more important functions in lists, but they will be defined after learning about additional functional operations in Haskell.

Table 3. Common list operations

Concatenate two lists	$list1 ++ list2$
Prepend $element$ to the $list$	$element : list$
Get the element at this position	$list !! position$
Merging two lists into lists of tuples, total length will be length of the shortest list	$zip list1 list2$
Get the first element	$head list$
Get the last element	$last list$
Get list but without the first element	$tail list$
Get list but without the last element	$init list$
Get a size of $list$	$length list$
Skip n first elements of $list$ or empty list if n is bigger than size of $list$	$drop n list$
Take n first elements of $list$ or unmodified $list$ if n is bigger than size of $list$	$take n list$

To modify a list or any other data structure it is required to construct a new one. It sounds limiting but recursive data structures may be more suitable in tasks that involves the recursion itself (Braithwaite 2017). Because it is not possible to change them in place; the creation of a new data structure is required. Some basic functions have been already defined, but for example removing one element in position n may be performed with the following functions:

$$removeAt\ n\ list = take\ n\ list ++ drop\ (n + 1)\ list \quad (65)$$

Lists are so powerful in Haskell that there is special syntax to create lists with the definition of simple rules that are called list comprehensions. It looks and works like set comprehensions in classical mathematics.

Example. Squares of the first 10 natural number as set and list comprehensions

$$\{x^2 \mid x \in \mathbb{N}, x \leq 10\} \quad (66)$$

$$[x^2 \mid x \leftarrow [1..10]] \quad (67)$$

The general syntax of list comprehensions is the following:

$$\begin{aligned}
 & [\langle function \rangle \mid \langle variables \rangle [, \langle predicates \rangle]] & (68) \\
 & \langle variables \rangle = variable \leftarrow list [, \langle another variable \rangle] \\
 & \langle predicates \rangle = predicate [, \langle predicate \rangle]
 \end{aligned}$$

Another way to create lists are ranges. The various ways of creating them are shown in Table 4.

Table 4. List generators

Returns a list of x with the length n	$replicate\ n\ x$
Returns a list with all elements from $start$ to end with the increment of 1	$[start..end]$
Returns an infinite list from $start$ and the increment of 1	$[start..]$
Returns a list with all elements from $start$ to end with the increment of $step$	$[start, (start + step)..end]$
Returns an infinite list from $start$ and the increment of $step$	$[start, (start + step)..]$

The second important data structure in Haskell are tuples. Tuples are relatively short as they cannot be infinite, but may mix any types together. The typical pattern is to use tuples when a function should return a concrete amount of outputs. Tuples don't have much helpful methods. It has only two methods which work only for two-element tuples: fst and snd which return the first or second element of tuple accordingly, but only work on two-element tuples as seen in Equations 69 and 70.

$$fst\ (a, b) = a \quad (69)$$

$$snd\ (a, b) = b \quad (70)$$

To get values from tuples with more elements the following syntax may be used:

$$(x, y, z) = tuple \quad (71)$$

If some function returns a tuple with three elements, this construction allows unrolling it into three variables. This syntax may be expanded to as many elements as needed.

Haskell doesn't have any other internal data structures. Before creating new data structures, types should be discussed first.

3.3 Types and functions

Haskell is a strongly typed language, i.e. every variable knows its type at the compile time. Signatures of functions in Haskell resemble ones in Typed Lambda Calculus but with added complexity and control. Signature of functions in Lambda Calculus part was already written in the Haskell notation to make the transition seamless. Declaring functions in Haskell does not require signatures but it's considered a good practice and usually helps the programmer himself.

An important feature of types in Haskell is type inference, the compiler can deduct which type should be the variable. Types may be defined using the following syntax:

$$\begin{aligned} \text{data } \langle \text{name} \rangle [\langle \text{parameters} \rangle] &= \langle \text{constructor} \rangle [| \langle \text{constructor} \rangle] \dots \quad (72) \\ \langle \text{constructor} \rangle &= \langle \text{name} \rangle [\langle \text{arguments} \rangle] \end{aligned}$$

The type declaration should have unique name and from 1 to many constructors separated by |. Constructors are functions which takes from 0 to many arguments. Parameters will be discussed a little bit later.

Example. The possible three-constructor type

$$\text{data Shape} = \text{Circle} | \text{Square} | \text{Rhombus} \quad (73)$$

Constructor may be recursive which is important feature for creating new data structures. One of the most popular data structures are trees. A common one is a binary tree which is a tree with maximum two children. A binary tree consists of a node and each node either empty or contains value and links to the left and right branches which are also trees. It can be defined as in the following example.

Example. The simple binary tree type

$$\text{data Tree } a = \text{Empty} | \text{Node } a (\text{Tree } a) (\text{Tree } a) \text{ deriving (Show)} \quad (74)$$

Types can be defined to implement specific classes. Type classes as name imply classify types into categories that have specific functionality, they are like Java

interfaces, e.g. *Eq* type class means that type can be checked for equivalence using $(==)$ and its counterpart $(/=)$. Type classes always start with a capital letter and has the following general form:

$$\begin{aligned} & \textit{class} \langle \textit{class name} \rangle \textit{ variable where} & (75) \\ & \quad \langle \textit{function signatures} \rangle \\ & \quad [\langle \textit{default implementation of functions} \rangle] \end{aligned}$$

Example. Simplified *Eq* type class definition

$$\begin{aligned} & \textit{class Eq a where} & (76) \\ & \quad (==) :: a \rightarrow a \rightarrow \textit{Bool} \\ & \quad (/=) :: a \rightarrow a \rightarrow \textit{Bool} \end{aligned}$$

When defining function signatures, it is possible to restrict arguments to specific classes, i.e. any type that belongs to that class can be used. The syntax of writing function signatures is the same as it was defined in Lambda Calculus and adding idea of class restrictions leads to the following:

$$\langle \textit{function name} \rangle :: [\langle \textit{class restrictions} \rangle \Rightarrow] \langle \textit{inputs and output} \rangle \quad (77)$$

Example. The function with two inputs of *Eq* class and return value of *Num* class

$$f :: (\textit{Eq} a, \textit{Num} b) \Rightarrow a \rightarrow a \rightarrow b \quad (78)$$

Having definitions of types and classes, it is possible to make a type an instance of the class. It has the following syntax:

$$\begin{aligned} & \textit{instance} \langle \textit{class name} \rangle \langle \textit{type name} \rangle \textit{ where} & (79) \\ & \quad \langle \textit{definition of functions} \rangle \end{aligned}$$

Example. The declaration of *Shape* type as an instance of *Eq* type class

$$\begin{aligned} & \textit{instance Eq Shape where} & (80) \\ & \quad \textit{Circle} == \textit{Circle} = \textit{True} \\ & \quad \textit{Square} == \textit{Square} = \textit{True} \\ & \quad \textit{Rhombus} == \textit{Rhombus} = \textit{True} \\ & \quad _ == _ = \textit{False} \end{aligned}$$

It should be noted that it is possible to use type constructors inside pattern matching. We defined our function the way we did it in Equation 49, if any of three cases occur then shapes are indeed the same and answer is *True*, otherwise it is *False*. Though, this example is illustrative, it is unnecessary to define such functions with obvious implementations and this type may be written as deriving *Eq* type class. The derivation of classes by types has the following general syntax:

$$\langle \text{type declaration} \rangle \text{ deriving } (\langle \text{class name} \rangle [\langle \text{class name} \rangle] \dots) \quad (81)$$

Our *Shape* also cannot be shown as string and thus it cannot be printed to the console. Function *show* is defined in *Show* type class is doing exactly that behavior and our class may be extended to derive this behavior also.

Example. *Shape* type deriving *Eq* and *Show* type classes

$$\text{data Shape} = \text{Circle} \mid \text{Square} \mid \text{Rhombus} \text{ deriving } (\text{Eq}, \text{Show}) \quad (82)$$

To continue our immersion into types let's create the type that can be parametrized. Parameters are the types that can be hidden inside another type. If the type has only one constructor, it is good practice to name it after the type.

Example. The simplest parametrized type

$$\text{data Box } a = \text{Box } a \quad (83)$$

Now it is possible to assign some variable inside *Box*.

Example. *Box* type with different arguments

$$\text{Box } 'a' :: \text{Box Char} \quad (84)$$

$$\text{Box } 4 :: \text{Box Int} \quad (85)$$

In functions it is possible to declare that any *Box* will work not with some specific internals, though it is possible to define it if needed. In signatures there can be only concrete types. A concrete type is any type that run a type constructor with all parameters filled, in case of zero parameters it is always concrete type, i.e. *Box 4* is a concrete type, but *Box* is a type constructor.

Example. Valid and invalid function signatures

$$\text{takeFromAnyBox} :: \text{Box } a \rightarrow a \quad (86)$$

$$\text{functionWithIntegerBox} :: a \rightarrow \text{Box Int} \quad (87)$$

$$\text{messUpWithBoxes} :: \text{Box} \rightarrow a \quad (88)$$

As it was mentioned it is possible to do pattern matching of type constructors with arguments in functions, i.e. it is possible to peak into insides of types and process them accordingly.

Example. Function that says if there is 42 inside *Box* type

$$\text{isItAnswer} :: \text{Box Int} \rightarrow \text{String} \quad (89)$$

$$\text{isItAnswer (Box 42)} = \text{"The answer is here"}$$

$$\text{isItAnswer } _ = \text{"Continue searching"}$$

Helpful feature of Haskell types is a possibility to create synonyms for types in the following way:

$$\text{type } \langle \text{synonym name} \rangle = \langle \text{original type name} \rangle \quad (90)$$

The keyword *type* is very misleading, it is only making a type synonym and not declaring new type. The best example when type synonym may be helpful is *String*. *String* is a list of characters then the type synonym for that is defined as the following:

$$\text{type String} = [\text{Char}] \quad (91)$$

Type synonyms are useful for the clearer description of a function. E.g. we want to write a program that will operate with names and prices of products represented by strings and doubles accordingly.

Example. Type synonyms for easier understanding of code

$$\text{type ProductName} = \text{String} \quad (92)$$

$$\text{type ProductPrice} = \text{Double} \quad (93)$$

Let's return to our *Box*, right now it is just a wrapper around anything and is good only for the illustrative purpose. But by just adding another constructor and

renaming will create one of the most important types in Haskell, *Maybe*, which has the following definition:

$$\text{data Maybe } a = \text{Nothing} \mid \text{Just } a \quad (94)$$

This type has the great power of holding value or not, this is the Haskell way of dealing with exceptions. Every function that can fail will return either *Just value* in case of a success or *Nothing* in case of a failure. Let's create a function that take numerical *Maybe* type, i.e. $(\text{Num } a) \Rightarrow \text{Maybe } a$, and try to multiply it by 2.

Example. The function that tries to multiply *Maybe* value by 2.

$$\text{maybeMultiplyBy2} :: (\text{Num } a) \Rightarrow \text{Maybe } a \rightarrow \text{Maybe } a \quad (95)$$

$$\text{maybeMultiplyBy2 Nothing} = \text{Nothing}$$

$$\text{maybeMultiplyBy2 (Just } a) = \text{Just } (a * 2)$$

Now our program will never crash if we'll supply *Maybe* value parsed by other function that could fail. It is easy to generalize multiplication and use two *Maybe* and try to multiply them. If any of them is *Nothing* then answer should be *Nothing* otherwise the multiplication of the insides.

Example. The function that try to multiply two *Maybe* values with numbers inside

$$\text{maybeMultiply} :: (\text{Num } a) \Rightarrow \text{Maybe } a \rightarrow \text{Maybe } a \rightarrow \text{Maybe } a \quad (96)$$

$$\text{maybeMultiply Nothing } _ = \text{Nothing}$$

$$\text{maybeMultiply } _ \text{ Nothing} = \text{Nothing}$$

$$\text{maybeMultiply (Just } a) (\text{Just } b) = \text{Just } (a * b)$$

Last thing that should be said is how to extract values from types. First, type that will have values extracted should be defined.

Example. *Color* type that will be used for arguments extraction

$$\text{data Color} = \text{Color Double Double Double} \quad (97)$$

Now, let's define functions to extract these values, like we have red, green and blue values.

Example. Functions to get arguments from *Color* type

$$red :: Color \rightarrow Double \quad (98)$$

$$red (Color red _ _) = red$$

$$green :: Color \rightarrow Double \quad (99)$$

$$green (Color _ green _) = green$$

$$blue :: Color \rightarrow Double \quad (100)$$

$$blue (Color _ _ blue) = blue$$

It is working, and it is possible to call *green (ourColor)* and get the *green* component of the type. But there is simple and preferable way called record syntax. It allows to give names to arguments that will automatically create functions to get them. Also, it allows to create type with setting arguments by names and in any order.

Example. The definition and creating *Color* type using the record syntax

$$data Color = Color \{red :: Double, green :: Double, blue :: Double\} \quad (101)$$

$$Color \{green = 2, red = 1, blue = 3\} \quad (102)$$

Haskell is left associative just like Lambda Calculus, i.e. the left-most function will try to take all arguments it encounters. But Haskell also has support for the infix functions. To make any binary function infix it should be written inside backticks.

Example. Usage of Equation 96 in infix manner

$$Just 3 \text{ `maybeMultiple` } Just 4 = Just 12 \quad (103)$$

Functions that consist only of special characters like *+*, ***, */*, *==*, *.*, *\$* are considered infix by default. To make such function prefix or to refer to it parentheses are required (*+*), (*==*), etc.

Example. Using multiplication as prefix function

$$(*) 3 4 = 12 \quad (104)$$

One of such infix functions is the application function `$`. The application function has the same form as Equation 7 and signature as Equation 39. It will not run until the right part is calculated completely. In general, `$` may be replaced with parentheses that start here and go until the end of expression.

Example. Differences between results with `$`, without it and with parentheses

$$\text{succ } 2 * 3 = 9 \quad (105)$$

$$\text{succ } (2 * 3) = 7 \quad (106)$$

$$\text{succ } \$ 2 * 3 = 7 \quad (107)$$

In the first case `succ 2` was evaluated first and $3 * 3 = 9$. In the second case $(2 * 3)$ evaluated first and then `succ 6 = 7`. In third case `succ` encounters `$` and lets everything after it to be evaluated first, $2 * 3 = 6$ and `succ $ 6 = succ 6 = 7`.

`$` has such property, because it is possible to set a fixity for the infix functions in Haskell. Functions may be set precedence from 0 to 9, where function `+` has 6, `*` has 7, but `$` has 0 and always evaluated the very last. Another point that the fixity declaration allows to set is left, right or no associativity, which sets in which order functions with the same precedence will be evaluated. The following construction is used to define fixity:

$$\langle \text{fixity} \rangle \langle \text{precedence} \rangle \langle \text{function name} \rangle [\langle \text{function name} \rangle] \dots \quad (108)$$

$$\langle \text{fixity} \rangle = \text{infix or infixl or infixr}$$

As an example, let's define a function with different fixity than usual. The simplest operations to notice are addition with multiplication.

Example. Addition that will be evaluated before multiplication

$$\text{infixl } 8 \text{ +++} \quad (109)$$

$$\text{(+++)} :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$$

$$x \text{ +++ } y = x + y$$

$$2 \text{ +++ } 3 * 4 = 5 * 4 = 20 \quad (110)$$

$$2 * 3 \text{ +++ } 4 * 5 = 2 * 7 * 5 = 70 \quad (111)$$

It should be noted that functions with the same precedence, but a different associativity must not be used inside one expression, as it is undefined in which order to execute them and an error will occur. The same will happen if two functions without associativity will be used in the same expression.

Another function that is helpful in functional programming for defining the order of execution is the function composition. Function composition is applying a function on the result of another one. Both functions should take one argument. Ways of defining it in math and Haskell can be seen in Equations 112 and 113.

$$f(g(x)) = (f \circ g)(x) \quad (112)$$

$$f(g\ x) = (f \cdot g)\ x \quad (113)$$

Function composition has the following signature and definition in Haskell:

$$(\cdot) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c) \quad (114)$$

$$f \cdot g = \lambda x \rightarrow f(g\ x)$$

Now we can create a new function using function composition. E.g. a function that will double an argument and then subtract 2. Ways to define it with and without the function composition may be seen in the following example.

Example. The function that doubles an argument and then subtract two from it

$$f\ x = \text{subTwo}\ (\text{double}\ x) \quad (115)$$

$$f\ x = (\text{subTwo} \cdot \text{double})\ x \quad (116)$$

Although it works, there is another preferred method. Instead of applying a function we can just define a function, i.e. we can omit an argument because when function will approach some argument, it will apply it to it.

Example. Previous example written in point-free style

$$f = \text{subTwo} \cdot \text{double} \quad (117)$$

It is widely encouraged to make all functions in Haskell like this, if possible. It is called point-free style. The name can be confusing, but it originates from the topology and the choice of operator (\cdot) in Haskell is just an unfortunate coincidence. (Haskell 2011.)

Officially all Haskell functions may take only one argument. But still we saw functions that disobey that rule like power function in Equation 50. Haskell has a special internal operation, which was already mentioned in Lambda Calculus part called currying, which is also named after the logician Haskell Curry. Currying is like a one-step beta-reduction in Lambda Calculus. It is a process of forming a new function by taking the first argument and then in case of several arguments this newly built function is taking it as an argument. Formally, it may be defined as in Equation 118. A function that takes two arguments is the function with a already inside that takes b as an argument.

$$f\ a\ b = (f\ a)\ b \quad (118)$$

Example. The maximum function and how it can be seen with currying

$$\mathit{max}\ a\ b = \mathit{if}\ a > b\ \mathit{then}\ a\ \mathit{else}\ b \quad (119)$$

$$\mathit{max} :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow a \rightarrow a$$

$$\mathit{max}\ 5\ b = \mathit{if}\ 5 > b\ \mathit{then}\ 5\ \mathit{else}\ b \quad (120)$$

$$\mathit{max}\ 5 :: (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow a \rightarrow a$$

If only one argument is passed it will return a function with a partial application and it will become a function that returns 5 or any bigger number. Any function goes through such process and the partial application will be helpful after learning about special operations including mapping.

3.4 Additional functional operations

There are three additional functional operations: mapping, filtering and folding. These operations are possible only due to fact that in Haskell functions are the first-order objects, i.e. they can be passed like a variable. Mapping is the process of applying a specific function to some data structure. Usual *map* function works only for lists. The way of extending such functionality to the other types will be discussed in the next subsection. *map* has the following signature:

$$\mathit{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \quad (121)$$

It takes a list and function that takes one input which must be the same type as the elements in a list and return a list with the same size with the elements of the

same or different type. Writing more explicitly, the result of mapping will be the following:

$$\text{map } f [a_0, a_1 \dots a_n] = [f a_0, f a_1 \dots f a_n] \quad (122)$$

The implementation would look as follows:

$$\text{map } _ [] = [] \quad (123)$$

$$\text{map } f (x:xs) = (f x) : (\text{map } f xs)$$

Example. The mapping of self-multiplication over a list

$$\text{map } (\backslash x \rightarrow x * x) [1,2,3,4] = [1,4,9,16] \quad (124)$$

Filtering is a process of removing elements that do not follow a predicate. It returns a list whose size will be between 0 and the initial size where the elements don't change their types or values. It has the following signature:

$$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \quad (125)$$

The possible implementation will be the following:

$$\text{filter } _ [] = [] \quad (126)$$

$$\text{filter } f x:xs$$

$$| f x = x : (\text{filter } f xs)$$

$$| \text{otherwise} = \text{filter } f xs$$

Example. Getting all the elements bigger than 2

$$\text{filter } (\backslash x \rightarrow x > 2) [1,2,3,4] = [3,4] \quad (127)$$

Folding is the process of generating one value from a list also sometimes referred to as reducing. Folding requires a function that takes two arguments, a starting value and a list. Folding is the most powerful function of functional programming. There are two types of folding: folding from the left *foldl* and from the right *foldr*. Equations 128 and 129 show the simplified signatures.

$$\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \quad (128)$$

$$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \quad (129)$$

Folding may be calling a function in the infix manner as follows:

$$\text{infixl } f \quad (130)$$

$$\text{start } \backslash f \ a_0 \ \backslash f \ a_1 \ \backslash f \ \dots \ \backslash f \ a_n$$

$$\begin{aligned} & \text{infixr } f && (131) \\ & a_0 \text{ `f` } a_1 \text{ `f` } \dots \text{ `f` } a_n \text{ `f` } \textit{start} \end{aligned}$$

It should be noted that the left folding will call a function next time only when the previous call is evaluated, when the right folding makes all the calls first and only then starts to evaluate.

If a function for folding is a lambda function, then arguments are usually called *acc* and *x*, to denote the accumulator and the current value accordingly. Folding applies the function between an accumulator and the values of the list one by one until the list is over.

Example. Summing all elements of list

$$\text{foldl } (+) 0 [1,2,3,4] = 10 \quad (132)$$

Sometimes, folding doesn't change the type of values, and it is possible to use the first value of list as the starting value. For that there is shorthand functions *foldl1* and *foldr1*.

Example. Summing a list with the starting value being the first element of the list

$$\text{foldl1 } (+) [1,2,3,4] = 10 \quad (133)$$

Let's return to our *Tree* and define a function to add an element to it. If the current value is smaller than the value of the current node, then we put it to the left. If it is bigger, then to the right. In case of equality return just current tree. It should be noted that this function does not create an optimal tree.

Example. The function to add an element to our *Tree* type

$$\begin{aligned} & \text{addToTree} :: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Tree } a && (134) \\ & \text{addToTree } x \text{ Empty} = \text{Node } x \text{ Empty Empty} \\ & \text{addToTree } x (\text{Node } \textit{value} \textit{left} \textit{right}) \\ & \quad | x == \textit{value} = \textit{value} \text{ Node } \textit{left} \textit{right} \\ & \quad | x < \textit{value} = \text{Node } \textit{value} (\text{addToTree } x \textit{left}) \textit{right} \\ & \quad | x > \textit{value} = \text{Node } \textit{value} \textit{left} (\text{addToTree } x \textit{right}) \end{aligned}$$

Now this function can utilize the existing tree and add an element, which in fact mean that it will construct new tree as the variables are immutable. By using folding it is possible to utilize this function to create a tree from a list like the following:

$$\text{foldr addToTree Empty list} \quad (135)$$

But, it also possible to utilize left folding even if *addToTree* has a wrong signature. The function *flip* which is defined in Equation 136 can help to change order of arguments.

$$\text{flip } f \ x \ y = f \ y \ x \quad (136)$$

And instead a binary tree may be created in the following manner:

$$\text{foldl (flip addToTree) Empty list} \quad (137)$$

As already mentioned, lists are a powerful and important tool in Haskell. There are several functional ways of interacting with lists shown in Table 5.

Table 5. Functional list operations

Skips elements while predicate <i>f</i> returns true and returns all further elements	<i>dropWhile f list</i>
Returns elements of a list while predicate <i>f</i> returns true, then skips all the other elements	<i>takeWhile f list</i>
Element-wise application of function <i>f</i>	<i>zipWith f list1 list2</i>
Returns true if, predicate <i>f</i> returns true on at least one element	<i>any f list</i>
Returns true if, predicate <i>f</i> returns true for all elements	<i>all f list</i>

Having so many functional tools, let's use them to calculate some examples.

Example. Various functional operations

$$\text{map (map } (\lambda x \rightarrow x * x)) \ [1,2,3], [4,5,6]] = [[1,4,9], [16,25,36]] \quad (138)$$

$$\text{filter } (\lambda x \rightarrow 100 \ `mod` \ x == 0) \ [1..100] = \text{whole divisors of 100} \quad (139)$$

$$\text{takeWhile } (\lambda x \rightarrow x \ `elem` "aeyuio") \ \text{string} = \text{first vowels of string} \quad (140)$$

$$\text{zipWith } (^) \ [2,2..] \ [0..] = \text{infinite list with powers of two} \quad (141)$$

3.5 Monads

Monads are the most core concepts of Haskell. Functional programming is mostly about composing functions and monads allow changing the way functions are composed (Boyer 2014). Before speaking about monads, *Functor* should be discussed first. Both ideas come from category theory which will not be covered in this thesis.

Functor type class describe how to apply functions to some type. It has the following definition without default implementation for *fmap*:

$$\begin{aligned} \text{class Functor } f \text{ where} & \quad (142) \\ \text{fmap} & :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \end{aligned}$$

fmap is the more general version of *map* and for the lists they produce the same results. *fmap* signature may resemble the application function signature seen in Equation 38. The important feature of that class definition is that *f* that we will make an instance of *Functor* type class is not a concrete type. It can be seen by lines *f a* and *f b* which imply that *f* should take some arguments, and thus, it is not a concrete type but a type constructor. This means that we can make our *Box* introduced in Equation 83 an instance of this type class.

Example. *Box* type can be an instance of *Functor* type class

$$\begin{aligned} \text{instance Functor Box where} & \quad (143) \\ \text{fmap } f \text{ (Box } a) & = \text{Box } (f\ a) \end{aligned}$$

Now it is possible to apply a function to the internals of *Box* and get back the result still packed into *Box*.

Example. Mapping of *succ* function upon *Box* object

$$\text{fmap succ (Box 4) = Box 5} \quad (144)$$

As previously, let's expand it to *Maybe* type as this is one of the most important types in Haskell and its being *Functor* is helpful. It is easy to see that the application of function upon *Maybe* is either an application of a function with internals as argument or *Nothing*, if there is no value. It is very similar to *Box* but

there is fallback strategy in *Maybe*. *Maybe* defined as an instance of *Functor* in the following way:

$$\text{instance Functor Maybe where} \quad (145)$$

$$fmap f (Just a) = Just (f a)$$

$$fmap f Nothing = Nothing$$

The important thing that should not be overseen when making something a *Functor* are *Functor* laws. They are not implied by Haskell, but when somebody would use something as *Functor* they would expect a specific behavior to be followed. *Functor* laws have the following formal definition:

$$fmap id a = id a \quad (146)$$

$$fmap (f . g) a = fmap f (fmap g a) \quad (147)$$

The first law states that there should be no difference between mapping identity and applying identity, i.e. mapping doesn't have side-effects. The second law states that there should be no difference between mapping function composition or mapping functions one by one. These laws are important as in functional programming pure functions must return the same results with the same input, and if laws are not obeyed it can be wrong and thus unreliable.

To uncover full potential of *Functor* let's make our *Tree* introduced in Equation 74 an instance of it. How do we apply some function to all elements inside *Tree*? First, we apply a function to the value of the current node and then apply a function to both the left and right trees, which again will either a call function on current value and branches. If branch is *Empty*, it will return *Empty* then.

Example. *Tree* type can also be an instance of *Functor* type class

$$\text{instance Functor Tree where} \quad (148)$$

$$fmap f Empty = Empty$$

$$fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)$$

Because of currying it is possible to use functions that requires more than one argument with *fmap*. In Haskell a and $(a \rightarrow b)$ are both valid objects, i.e. a list of functions is as natural as a list of integers. Let's imagine, some $f :: a \rightarrow b \rightarrow c$, which and $f a :: (b \rightarrow c)$ which may be stored inside a list in the following way:

$$fmap f [a_0, a_1 \dots a_n] = [(f a_0), (f a_1) \dots (f a_n)] \quad (149)$$

Example. A list of multiplications and its signature

$$fmap (*) [4,5] = [(* 4), (* 5)] \quad (150)$$

$$fmap (*) [4,5] :: [Int \to Int] \quad (151)$$

Now we need to access functions inside a list, the naïve way is to iterate through this list and apply it using a lambda function.

Example. Applying a list of functions

$$fmap (\f \to f 9) a = [36,45] \quad (152)$$

$$\text{where } a = fmap (*) [4,5]$$

We have just used a list of functions over a single value, what if we want to run that function over a list of values? Then more mapping is required.

Example. Applying a list of functions on a list of arguments

$$fmap (\f \to fmap f [9,10]) a = [[36,40], [45,50]] \quad (153)$$

$$\text{where } a = fmap (*) [4,5]$$

But this is barely readable! There is enhanced way called *Applicative Functor*. It is a type class with the following definition:

$$\text{class (Functor } f) \Rightarrow \text{Applicative } f \text{ where} \quad (154)$$

$$\text{pure} :: a \to f a$$

$$\langle * \rangle :: f (a \to b) \to f a \to f b$$

For something to be *Applicative Functor* it must be *Functor* first. *pure* function puts a value into the minimal context, it means that the value will be inside type *f* but it is still the same value. $\langle * \rangle$ looks like *fmap* and its implementation usually uses it. *Box* can be easily an instance of *Applicative Functor*.

Example. *Box* is an instance of *Applicative Functor*

$$\text{instance Applicative Box where} \quad (155)$$

$$\text{pure} = \text{Box}$$

$$\text{Box } f \langle * \rangle x = \text{fmap } f x$$

pure function puts a value into the minimal context and the result depends into which context it should be put, i.e. *pure* 9 may have as many interpretations as there are instances of *Applicative Functor* class. It is possible to enforce to which type function *pure* will use like shown in Equation 156.

Example. Using *pure* to enforce numeric *Box*

$$\text{pure } 4 :: \text{Num } a \Rightarrow \text{Box } a = \text{Box } 4 \quad (156)$$

To show how $\langle * \rangle$ works Equation 153 will be refined into Equation 157. It perfectly replaces the lambda function that we had and make the order of arguments more readable. Though, there is a little difference of list being flattened which happens due to the list's implementation of $\langle * \rangle$.

Example. Applying a list of function on a list of arguments using $\langle * \rangle$

$$\begin{aligned} a \langle * \rangle [9,10] &= [36,40,45,50] \\ \text{where } a &= \text{fmap } (*) [4,5] \end{aligned} \quad (157)$$

But a list of functions is still created old way, let's try to use *pure* function.

Example. Creating a list of function with the help of *pure* function

$$\text{pure } (*) \langle * \rangle [4,5] = [(* 4), (* 5)] \quad (158)$$

Now, if we combine it with the previous example it will yield the following formula.

Example. Combining *pure* and $\langle * \rangle$

$$\text{pure } (*) \langle * \rangle [4,5] \langle * \rangle [9,10] = [36,40,45,50] \quad (159)$$

It is now more readable and what's more important it is scalable. Let's imagine a function *g* that takes more than one argument. Then the following form can be used to apply function to as much arguments as needed:

$$\text{pure } g \langle * \rangle x [\langle * \rangle y] \dots \quad (160)$$

And it still possible to improve it, instead of calling *pure g <*>* it is possible to write *g <\$>* where *f <\$> x = fmap f x*. It leads to the ultimate way to use *Applicative Functor*:

$$g \langle \$ \rangle x [\langle * \rangle y] \dots \quad (161)$$

Now it is possible to come to the monads. A type that we make an instance of *Monad* type class should be already an instance of *Applicative Functor*. Class definition has the following form:

$$\begin{aligned} \text{class } (\text{Applicative } m) \Rightarrow \text{Monad } m \text{ where} & \quad (162) \\ \text{return} &:: a \rightarrow m a \\ (\gg=) &:: m a \rightarrow (a \rightarrow m b) \rightarrow m b \\ (\gg) &:: m a \rightarrow m b \rightarrow m b \\ x \gg y &= x \gg= _ \rightarrow y \\ \text{fail} &:: \text{String} \rightarrow m a \\ \text{fail } msg &= \text{error } msg \end{aligned}$$

It is easy to see that *return* function is the same as *pure* function, it puts value into the minimal context. It should be noted that *return* is just the name of a function and don't have anything related with the return keyword in languages like Java. The second function $\gg=$ is called *bind* and is the main part of a monad. It takes a monadic value and a function that takes a normal value and returns a monadic one. Function \gg is a shortcut for taking two monadic values and return the second one. *fail* function is called when something is not right, the default implementation throws an error which is greatly discouraged and should be replaced with something more graceful like returning *Nothing*.

There are three monads where we have already used two of them widely: lists and our favorite *Maybe*, third is the most important monad is *IO* which will be discussed a little bit later.

Now we will define *Maybe* as an instance of *Monad* in Equation 163. Monads work best with types that can be in several states or have several values inside like tuples and making *Box* an instance of *Monad* serve little to none purpose. The minimal class definition requires to implement *return* and *bind* functions,

however, as I have already mentioned leaving the default implementation for *fail* is greatly discouraged. In *Maybe* some failure will just result in *Nothing*.

$$\begin{aligned} & \text{instance Monad Maybe where} && (163) \\ & \quad \text{return} = \text{Just} \\ & \quad \text{Nothing} \gg= \text{Nothing} \\ & \quad \text{Just } x \gg= f = f \ x \\ & \quad \text{fail } _ = \text{Nothing} \end{aligned}$$

It was the first part of working with monads – defining an instance. The second part requires an appropriate function to work with monads. Let's imagine, we play a game with adding integers to the sum and if the sum becomes too big then it is the end of our game. Our function should take a usual value and return a monadic one.

Example. The function that takes two values and check if the sum is too big

$$\begin{aligned} & \text{addToSum} :: (\text{Ord } a, \text{Num } a) \Rightarrow a \rightarrow a \rightarrow \text{Maybe } a && (164) \\ & \text{addToSum sum } x \\ & \quad | \text{newSum} \geq 10 = \text{Nothing} \\ & \quad | \text{otherwise} = \text{Just newSum} \\ & \quad \text{where newSum} = \text{sum} + x \end{aligned}$$

Now, we have two parts that can be combined to work together. Also, we should note that the function can work by itself, without using *bind*. The following results may be seen of calling function normal and monadic way.

Example. Using the function with and without *bind* operation

$$\text{addToSum } 3 \ 4 = \text{Just } 7 \quad (165)$$

$$\text{return } 4 \gg= \text{addToSum } 3 = \text{Just } 7 \quad (166)$$

It is easy to see that our function takes two values and return a monadic value of *Maybe*. But what if we want to apply function several times? Then we just call *bind* function as much times as needed because usual call doesn't let us do it.

Example. Chaining of *bind* operations

$$\text{return } 4 \gg= \text{addToSum } 3 \gg= \text{addToSum } 2 = \text{Just } 9 \quad (167)$$

This is the simple way of chaining calls of functions that work with monads. Now let's tinker arguments to make it return *Nothing*.

Example. Chains that will result in *Nothing*

$$\text{return } 4 \gg= \text{addToSum } 3 \gg= \text{addToSum } 11 = \text{Nothing} \quad (168)$$

$$\text{return } 4 \gg= \text{addToSum } 11 \gg= \text{addToSum } 2 = \text{Nothing} \quad (169)$$

$$\text{return } 11 \gg= \text{addToSum } 3 \gg= \text{addToSum } 2 = \text{Nothing} \quad (170)$$

We don't care when it fails, only whether it fails or not. Creating a failover function with signature $a \rightarrow \text{Maybe } b$ it is possible to chain operations and we don't need to check at each step whether the game is over or not.

There was one more function that we didn't use yet, \gg . It is possible to use it to create the 100% failure as only the second value will be returned.

Example. Using \gg to make whole chain return *Nothing*

$$\text{return } 4 \gg \text{Nothing} \gg= \text{addToSum } 2 = \text{Nothing} \quad (171)$$

So far, we've been writing everything inline and it becomes hard to read and write in case of a big number of calculations. Monads provide another feature called *do* notation. Let's rewrite our expression using it and show how compiler converts it.

Example. Rewriting the chain of operations into *do* notation

$$\text{playGame} = \text{do} \quad (172)$$

$$x \leftarrow \text{return } 4$$

$$y \leftarrow \text{addToSum } x \ 3$$

$$\text{addToSum } y \ 2$$

$$\text{return } 4 \gg= (\backslash x \rightarrow \text{addToSum } x \ 3 \gg= (\backslash y \rightarrow \text{addToSum } y \ 2)) \quad (173)$$

$$\text{playGame} = \text{Just } 9 \quad (174)$$

So, what can we see? At the same time variables behave like normal and monadic values. *return 4* returns monadic value but *addToSum x 3* should use normal value it means that \leftarrow inside *do* has the power of uncovering a monadic value and assigning it to the variable. The important feature is that type of a function is the last action performed, that's why we don't bind last action with \leftarrow , as it will be done automatically for us. Let's try to rewrite some example with failure inside it.

Example. A fail game with *do* notation

$$\text{playFailGame} = \text{do} \tag{175}$$

$$x \leftarrow \text{return } 4$$

$$y \leftarrow \text{addToSum } x \ 11$$

$$\text{addToSum } y \ 2$$

$$\text{playFailGame} = \text{Nothing} \tag{176}$$

It looks like imperative programming but with the power of the implicit exception handling.

After looking on some examples of monads it worth mentioning that there are several laws that monads define. They are like *Functor* laws, are not implied by Haskell but everybody expects them to hold. In fact, we've already encountered the first monad law shown in Equations 165 and 166. It states that these two ways of calling a function must yield the same results. The second law states that binding of *return* function should not alter the input. The third law states that it should be no difference between chaining using *bind* function or using a usual application. Formally they may be defined in the following way:

$$\text{return } x \gg= f = f \ x \tag{177}$$

$$m \gg= \text{return} = m \tag{178}$$

$$m \gg= f \gg= g = m \gg= (\backslash x \rightarrow f \ x \gg= g) \tag{179}$$

The first law should hold true because that the way a function should apply a monadic value with a function that takes a non-monadic argument. The second law should hold true because *bind* must unfold *Monad* to a normal value and apply any function that must take a normal value and return monadic one. *return*

function should just put a normal value into the minimal monadic context. Summing these two facts gives to us that binding *return* function should unfold and instantly fold a monadic value back resulting in the zero changes. The third law is harder to see why it should hold true. Binding to the lambda function unfolds *m* and allows us to apply a function usual way to the inside *x* which is as we stated in the first monad law should be the same as binding. Then we use *bind* to apply the second function *g*.

Another important monad is *Writer*. It allows to utilize logging in Haskell. The idea is to have a tuple with the main object that is being processed and log messages that are appended to the log object. The usual pattern is the following: create a function that returns *Writer* type, inside the function call *tell* function to append something to the log and then call *return* with the answer.

Example. The function that make whole division by 2 and says what the input was

```
logDiviveByTwo :: Int → Writer String Int           (180)
logDivideByTwo x = do
  tell "Input was" ++ show x
  return x `div` 2
```

It is possible to make several calls to *tell* function and all of them will be appended to each other.

It's not necessary should be *String* to store logs. It can be any type that is an instance of *Monoid* type class. It has the following definition:

```
class Monoid m where                               (181)
  mempty :: m
  mappend :: m → m → m
  mconcat :: [m] → m
  mconcat = foldr mappend mempty
```

Monoid class define one binary associative operation and an identity element, e.g. numbers are monoids under addition which is binary associative operation

with the identity element of 0. Same goes for numbers under multiplication with the identity element of 1. There are three rules that can formally define it shown in Equations 182-184.

$$\text{mempty} \text{ `mappend` } x = x \quad (182)$$

$$x \text{ `mappend` } \text{mempty} = x \quad (183)$$

$$(x \text{ `mappend` } y) \text{ `mappend` } z = x \text{ `mappend` } (y \text{ `mappend` } z) \quad (184)$$

To make it easier to understand let's watch an example of addition. In means that $\text{mappend} = +$ and $\text{mempty} = 0$. Rewriting with substitution will lead to the following results:

$$0 + x = x \quad (185)$$

$$x + 0 = x \quad (186)$$

$$(x + y) + z = x + (y + z) \quad (187)$$

Now it is extremely easy to see that monoids mean. Same procedure may be performed for multiplication. One of the most important monoids is list with $\text{mempty} = []$ and $\text{mappend} = ++$. The last function that wasn't covered is mconcat . It has default implementation which folds using mappend between all elements and mempty . Usually this is good and there is no need in implementing it ourselves.

Now it is time to speak about the most important monad in the whole Haskell – IO . Haskell has strict rules and a usual code is not allowed to perform any IO operations. Main idea behind functional programming is functions that return same output by the same input, so-called pure functions. The reliability of the answer is what differentiate them from impure functions. To call impure functions, functions that can interact with a system should be inside an impure calculation itself. To make it work the very first main function should be impure and indeed it is. Let's introduce a simple IO function to be used in main :

$$\text{putStrLn} :: \text{String} \rightarrow IO () \quad (188)$$

We have IO which is monad but what does it wrap in current case? $()$ is a zero-element tuple with a type $()$ and there is only one such tuple. It is used to represent a dummy value that this is IO action without any kind of return value. Now we can create the most canonical program.

Example. Haskell's Hello World

```
main = putStrLn "Hello World!" (189)
```

Because *main* is monadic calculation it is possible to use *do* syntax.

Example. Hello World with *do* syntax

```
main = do (190)
  putStrLn "Hello"
  putStrLn "World!"
```

Let's make our program more interactive by requesting prompt from user by *getLine* which has the following signature:

```
getLine :: IO String (191)
```

It returns *IO* with *String* hidden inside it, \leftarrow can be used to retrieve normal value out of monad inside *do* block.

Example. A simple program that asks for user prompt

```
main = do (192)
  putStrLn "What is you name?"
  name ← getLine
  putStrLn ("Hello" ++ name)
```

4 CONFIGURING THE ENVIRONMENTS

To conduit the research, environments for writing, compiling and executing on Haskell, Java and Python are needed. Besides installing the basic compiling and execution environments some additional libraries and tools are needed to perform benchmarking on points given. It is hard to precisely measure CPU time, and elapsed time will be calculated instead. All work was done on Windows 7, 64-bit system but most steps should be similar for Linux and OS X systems.

4.1 Configuring Haskell

Haskell has several compilers with Glasgow Haskell Compiler (GHC) being the most popular one. It is distributed as part of Haskell Platform that also contains various helpful tools. Together they can be downloaded from Haskell official website (Haskell 2017a). For simpler and more Unix-like usage, Cygwin toolset with its own command prompt that supports Unix commands is downloaded from Cygwin official website (Cygwin 2017).

Some Integrated Development Environment (IDE) would be helpful but as our code will be compiled and ran through Cygwin, a text editor with Haskell plugin will suffice. Atom was chosen as one with immense support from the Haskell community. Atom is downloaded from Atom official website (GitHub Inc 2017). The Haskell plugin for Atom can be installed from Atom-Haskell official website (Atom-Haskell 2017). Opening the command prompt at the folder with Atom *apm* utility that can be found at the *installation path\bin* and running the following command will setup Haskell plugins for Atom:

```
apm install language-haskell ide-haskell ide-haskell-cabal      (193)
haskell-ghc-mod autocomplete-haskell
```

Then binary dependencies may be installed using *stack* that is part of Haskell Platform. The following commands should be executed in the command prompt:

```
stack install stylish-haskell                                  (194)
stack install ghc-mod
```

Programs are written with the *.hs* extension and will be compiled from the Cygwin command prompt. The flag *-O* is required for the compiler to perform optimization routines (Haskell 2017b). The complete command to compile with optimization is the following:

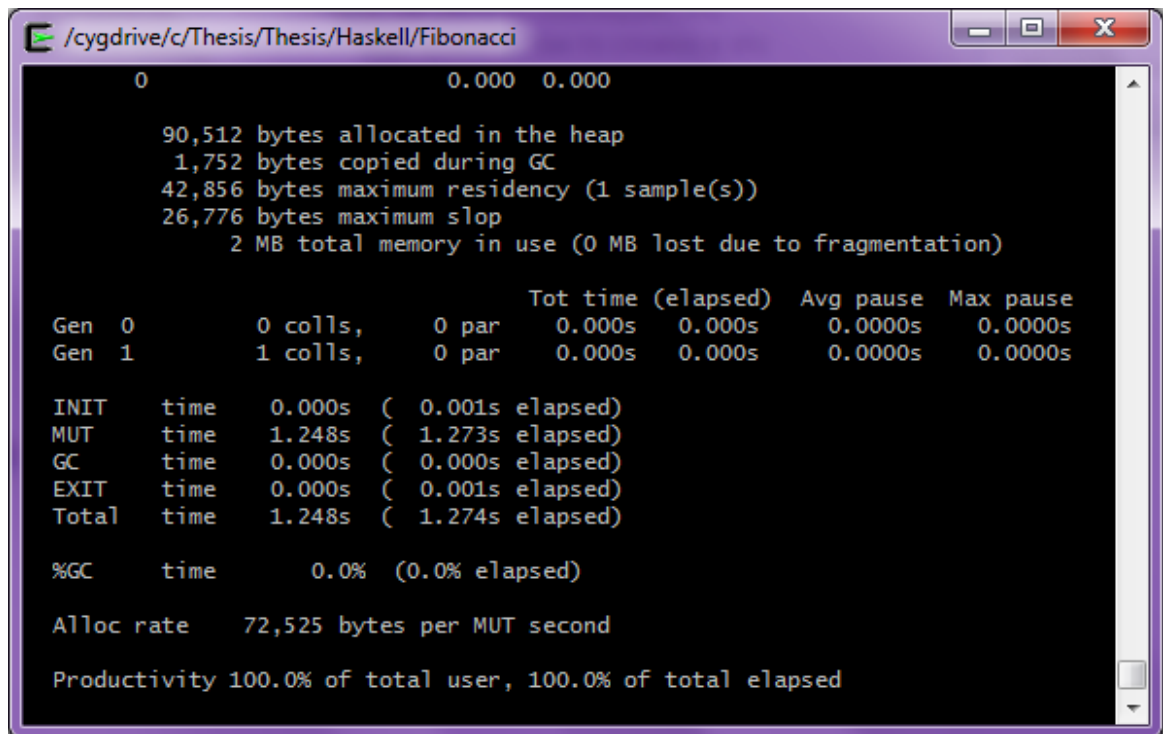
```
ghc -O <file name>                                           (195)
```

To get measurements we are interested program should be run with the *+RTS -S* flags (Haskell 2017b). With these flags, the measurement information will be send into the error stream. To redirect the error stream from the command

prompt to a file, the `2>file` syntax is used which is the standard Unix way. The complete command to run a program with measurements redirected to the file is the following:

```
./<file name> +RTS -S 2>output (196)
```

Figure 1 shows sample output without redirecting stream. The points of interest are the maximum residency and MUT which is the calculation time.



```

/cygdrive/c/Thesis/Thesis/Haskell/Fibonacci
0                               0.000 0.000

90,512 bytes allocated in the heap
 1,752 bytes copied during GC
42,856 bytes maximum residency (1 sample(s))
26,776 bytes maximum slop
 2 MB total memory in use (0 MB lost due to fragmentation)

                               Tot time (elapsed)  Avg pause  Max pause
Gen  0             0 colls,    0 par    0.000s   0.000s    0.0000s   0.0000s
Gen  1             1 colls,    0 par    0.000s   0.000s    0.0000s   0.0000s

INIT   time    0.000s ( 0.001s elapsed)
MUT    time    1.248s ( 1.273s elapsed)
GC     time    0.000s ( 0.000s elapsed)
EXIT   time    0.000s ( 0.001s elapsed)
Total  time    1.248s ( 1.274s elapsed)

%GC    time     0.0% (0.0% elapsed)

Alloc rate   72,525 bytes per MUT second

Productivity 100.0% of total user, 100.0% of total elapsed

```

Figure 1. Sample output of running a Haskell program with measurements

4.2 Configuring Java

First, latest Java Development Kit (JDK) is downloaded from Java official website (Oracle 2017). Java has good choice of IDEs and will stick to Eclipse which I've been using for very long time. It can be downloaded from Eclipse official website (The Eclipse Foundation 2017).

To measure time of execution, time difference with help of `System.nanoTime()` which will be used before will be executed before to store initial value and after main workload and difference between values will be used to determine CPU time (Nadeau 2008).

To find used memory it is not enough to get used memory after the main workload by using internal tools offered by *Runtime* class as garbage cleaner may be used during computation. Instead, to check the maximum used memory VisualVM tool will be used (Oracle 2016). It is distributed as part of JDK and can be found at *installation path\bin\jvisualvm.exe*. The heap graph is shown on Figure 2.

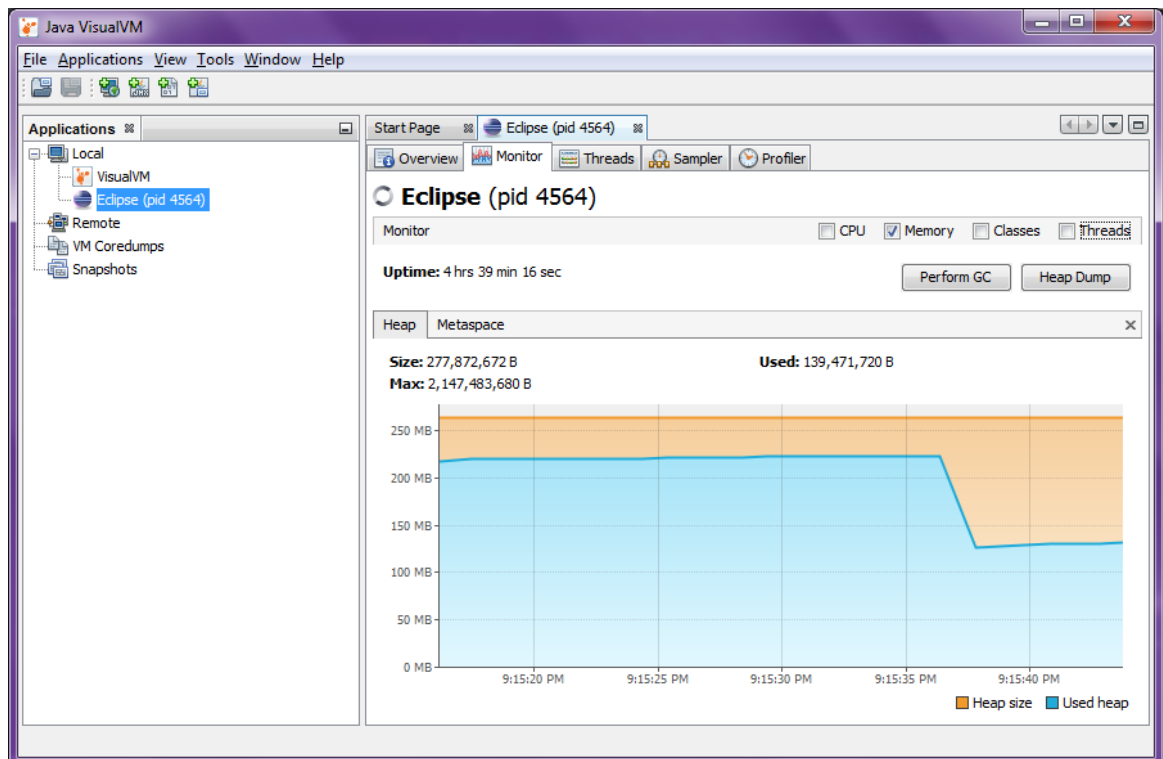


Figure 2. *VisualVM* screen with used memory graph

4.3 Configuring Python

Python has long going simultaneous release of version 2 and 3, version 3 was chosen as it is recommended. It can be downloaded from Python official website (Python Software Foundation 2017d).

As IDE Atom will be continued to use. For easier development Python Tools (Aquilina 2017) and Python Autocomplete Package (Sadovnychy 2017) are installed with the following command:

```
apm install python-tools autocomplete-python (197)
```

To make latter package work *jedi* Python package should be installed (Python Software Foundation 2017b) using *pip* in the following way:

```
pip install jedi (198)
```

To measure elapsed time, *time* library already included into standard Python installation. The following code snippet shows how the elapsed time will be measured:

```
import time (199)
startTime = time.time_clock()
function()
print(time.time_clock() - startTime)
```

psutil will be used to measure the peak memory during an execution (Python Software Foundation 2017c). It is not included in Python immediately but can be downloaded with Python package manager *pip* (PyPA 2016). To install *psutil* the following command should be executed in command prompt:

```
pip install psutil (200)
```

The following code snippet will show an example of using *psutil* to get the maximum memory used for the main thread:

```
import os (201)
import psutil
process = psutil.Process(os.getpid())
print(process.memory_info().peak_wset)
```

5 APPLICATION

The comparison of Haskell to other languages will be introduce with several rather small examples. Examples of features to test are additional functional operations (mapping, filtering, folding), pattern matching, cycles, etc. All source code of the comparison programs may be found at the thesis repository at my GitHub account (Prokopev 2017).

5.1 Summing test

First, some baseline performance for languages should be measured. One of the simplest tests is to run an empty cycle to see how much time is needed for that. But due to the nature of Haskell not having any cycles, and Python also not having usual cycles, the summing of numbers up to some n will be used. Based on tests $1e8$ is a value where all languages start to require noticeable time. Additionally, $1e9$ value will be used to test whether the time consumption will grow linearly or not.

One thing that should be considered is the size of an integer that a programming language can handle. Python's integer is unbounded. Haskell has two different ones: *Int* and *Integer* where the first is faster but bounded at 2^{63} . Java's *long* is bounded at same max value as Haskell's *Int*, and to work with bigger numbers *BigInteger* is required. Tests on Haskell and Java will be performed using both.

On Figures 3 and 4 summing in Haskell and Python are shown, which are defined like a sum of the all elements of a list. In Java the classical way with using a cycle is shown that can be seen on Figure 5. Also in Java the summing takes a Boolean argument to define whether use *long* or *BigInteger*.

```
summ :: (Integral a) => a -> a
summ n = sum [1..n]
```

Figure 3. Haskell summing list

```
def summ(n):
    return sum(range(n + 1))
```

Figure 4. Python summing list

```

static public BigInteger sumBigInteger(long n) {
    BigInteger answer = BigInteger.ZERO;

    for(long i = 1; i <= n; i++) {
        answer = answer.add(BigInteger.valueOf(i));
    }

    return answer;
}

```

Figure 5. Java summing with *BigInteger*

The results may be seen in Table 6. The results of the time consumption are the following: Java and Haskell are extremely fast if using small bounded version of integer values, but in case of working with numbers which exceeds 2^{63} their speed fades. Python that have only unbounded integer is much slower even compared to Haskell's unbounded one. Also, there is another implementation of Python called PyPy which I by this test is around 3 times faster than usual Python and developers state that it should be 7.5 times faster on average (The PyPy project 2017).

Table 6. Results of the summing test

	Time	Max memory
Haskell <i>Int</i> 1e8	0.078 s	42 KB
Haskell <i>Integer</i> 1e8	3.96 s	42 KB
Python 1e8	14.174 s	10.496 MB
Java <i>long</i> 1e8	0.089 s	1.16 MB
Java <i>BigInteger</i> 1e8	4.797 s	569.312 MB
Haskell <i>Int</i> 1e9	1.016 s	42 KB
Haskell <i>Integer</i> 1e9	40.258 s	42 KB
Python 1e9	2 m 14 s	10.613 MB
Java <i>long</i> 1e9	1.165 s	9.289 MB
Java <i>BigInteger</i> 1e9	46.285 s	496.17 MB

Memory footprint is slighter harder thing to interpret and some clarifications are needed. Haskell has extremely, and suspiciously low memory footprint which is happening due to the powerful optimization GHC provides. Python memory usage doesn't differ much from counting bigger sum which is positive sign. Java

memory usage on task with *long* and $1e8$ is such small because Java Virtual Machine (JVM) didn't have enough time to start all usual routines and thus such measured value is inaccurate. On the other hand, values that were received for *BigInteger* are also unrepresentative if we don't mention that garbage cleaner ran several times during execution which may be seen on Figure 6. Yet, giving JVM more allocated memory then it can reserve more memory as seen in Figure 7 where Java heap has 4 GB space.

Summary of first test: due to lazy evaluation and list ranges Haskell allows summing using a simple structure with small memory footprint. Java with *BigInteger*, on the other hand, adds unnecessary complexity and uses enormous amount of memory to be able to sum big numbers.

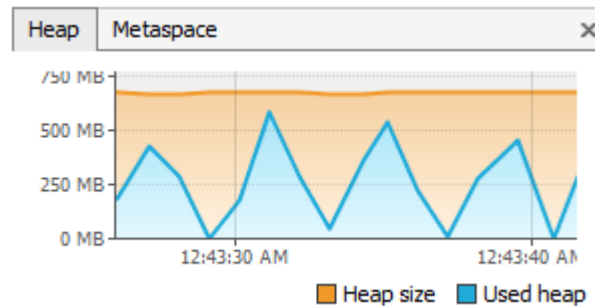


Figure 6. Java memory usage when using *BigInteger*

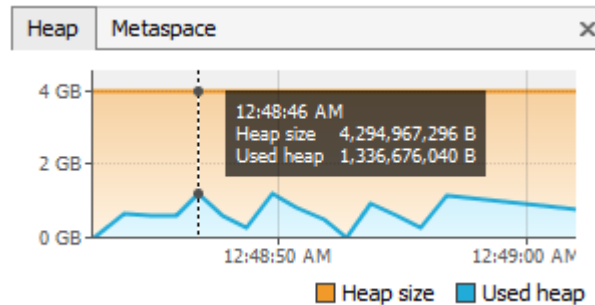


Figure 7. The *BigInteger* memory usage with an increased heap space

5.2 Recursion test

The second test will be recursive algorithm to find n-th Fibonacci number. Fibonacci sequence is typical example of algorithm that can be expressed using recursion and typical example of how to make simple algorithm run for too long.

Base case is getting number at 1 and 2 position which both are equal to 1, otherwise return sum of two previous, namely $(n - 1)$ and $(n - 2)$ numbers (MathWorld 2017). It should be noted that recursive implementation is extremely bad algorithmically-wise. In Haskell this algorithm can have implementation shown on Figure 8. Implementations in other languages will be similar, the main difference is that Haskell allows to use multi-declaration function which is just syntactic sugar for switch or if-else structures that can be implemented in two other languages. On Figure 8 and 9 it is easy to notice that the implementation of Fibonacci in Haskell and Python are not that different.

```
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Figure 8. The inefficient Fibonacci implementation in Haskell

```
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Figure 9. The inefficient Fibonacci implementation in Python

Table 7. Results of the Fibonacci test

	Time	Max memory
Haskell <i>fib</i> 40	1.383 s	42 KB
Python <i>fib</i> (40)	57.166 s	10.637 MB
Java <i>fib</i> (40)	0.553 s	1.16 MB
Haskell <i>fib</i> (50)	2 m 37 s	42 KB
Python <i>fib</i> (50)	58 m 20 s	11.91 MB
Java <i>fib</i> (50)	1 m 9 s	15.674 MB

The results are presented in Table 7. This test is just more complicated summing. The time consumption between Haskell and Java is not that big as it was in

Python, but previous test showed that Python's unbounded integer is not efficient for performing big number of operations. Haskell's memory footprint didn't change at all which is due to the amazing optimization. Python and Java memory usage don't go much behind their minimum requirement of about 10 MB each. Implementation of this test was again similar, and no big differences may be found.

The summary of the second test: Haskell still has amazingly low memory footprint, Python is not good for a big amount of summing which is performed not by *sum* function.

5.3 Derivatives list

As functional programming implies it should be good at working with mathematical functions. To check it, the program that will find simple derivatives will be written. Derivatives show how the functions value is changing by infinitesimal change in the input variable or variables (MathWorld 2017).

The following functions' derivatives will be implemented:

- Power
- Cosine
- Sine
- Natural logarithm
- Sum
- Subtraction
- Multiplication
- Division

The first step of writing this program is to define special structures upon which derive action will be defined. Then the derive function should be written. Also, the simplification function with some basic rules like $0 + x = x$ and $0 * x = 0$ was created. Lastly, it would be preferable to teach structures how to be printed into a human-readable format. An optional feature that would parse an input into our special structure world will not be implemented as being too complex.

Besides defining functions that were already mentioned constructors for numbers, number e and variable were defined. Because Haskell allows to use constructors of types inside pattern matching it is simple to define derivative. E.g. derivative of sum of anything is derivative of first argument plus derivative of second argument. More about derivatives may be Writing it mathematical and Haskell ways will yield the following equations:

$$(a + b)' = a' + b' \quad (202)$$

$$\text{derive } (\text{Sum } a \ b) = \text{Sum } (\text{derive } a) \ (\text{derive } b) \quad (203)$$

It is extremely simple yet powerful syntax that Haskell allows us to do. Implementations in Java and Python were done using OOP. The main difference is that in Haskell type was introduced and then functions were created that take argument of our type and return that same type. Also, it was made an instance of *Show* type class to allow conversion to human-readable String.

In Java and Python on the other hand we created class for each type constructor we have in Haskell and defined operations inside class. Additionally, Java required that all of them was implementing same interface which has name of function we need, namely functions *derive* and *simplify*. Haskell didn't require that as everything was of the same type and Python has dynamic typization which allows to use classes any way we like.

On Figure 10 the definition of *Derivable* type is shown which only derives *Eq* as it obvious how it should be implemented. Figure 11 and 12 shows one of the derivable class in Java and Python on example of *Sin* class. Figure 13 shows *Derivable* interface in Java that define two functions used by classes.

```

data Derivable a = Product (Derivable a) (Derivable a)
                  | Division (Derivable a) (Derivable a)
                  | Sum (Derivable a) (Derivable a)
                  | Sub (Derivable a) (Derivable a)
                  | Power (Derivable a) (Derivable a)
                  | Log (Derivable a)
                  | Cos (Derivable a)
                  | Sin (Derivable a)
                  | Neg (Derivable a)
                  | Const a
                  | E
                  | X
  deriving (Eq)

```

Figure 10. Haskell definition of *Derivable* type

```

class Sin(object):
    def __init__(self, a):
        self._a = a

    def derive(self):
        return Product(Cos(self._a), self._a.derive())

    def simplify(self):
        self._a = self._a.simplify()
        return self

    def __str__(self):
        return "Sin(" + str(self._a) + ")"

```

Figure 11. Python declaration of *Sin* class

```

public class Sin implements Derivable {
    public Sin(Derivable a) {
        this.a = a;
    }

    Derivable a;

    @Override
    public Derivable derive() {
        return new Product(new Cos(a), a.derive());
    }

    @Override
    public Derivable simplify() {
        a = a.simplify();
        return this;
    }

    @Override
    public String toString() {
        return "Sin(" + a + ")";
    }
}

```

Figure 12. Java declaration of *Sin* class

```

interface Derivable {
    public Derivable derive();
    public Derivable simplify();
}

```

Figure 13. Java *Derivable* interface

It is easy to see similarities and slight differences between Java and Python implementations. Definition of one class has constructor, two functions of interest namely `derive` and `simplify` and utility function that converts whole class to human-readable `String`. The main difference that Java is stricter like requirement of `@Override` annotations and requirement of the interface.

Table 8. Results of the derivatives test

	Time	Max memory
Haskell	< 0.001 s	42 KB
Python	0.0003 s	10.68 MB
Java	0.011 s	9 MB

Table 8 holds the results. The time consumption is extremely low in this test. Haskell spend so low time that there was not enough precision to measure it.

Python suddenly was much faster than Java probably due to the fact of using classes and Java requires more time to create ones. Memory consumption shows usual values which is just the runtime requirements.

Speaking about implementation this test was much simpler to write on Haskell. Amount of lines required are much less compared to other two languages. Haskell types has something in common with OOP definition of classes with additional possibility to use internal in the pattern matching.

The summary of this test: Haskell is amazing for writing a solution for tasks that may be expressed directly in mathematics. Also, it is faster and doesn't use much memory. Between Java and Python there are almost no differences.

5.4 Dijkstra test

One of the most important fields of mathematics is the graph theory. Besides how to store a graph in a computer it is important how to work with it. Due to Haskell's pure functional approach it is impossible to change variables after an assignment and because, as it was already stated many times, all data structures in Haskell are recursive. Combining these two facts leads us to the conclusion that to make a several operations upon some structure, each operation should be performed in its own recursive call, which is a tough challenge.

The Dijkstra algorithm has the following steps:

1. Set the shortest distance to starting node to 0.
2. Set the shortest distance to other nodes to infinity.
3. Take the current node and set the new shortest distance to the destination node, if the sum of distance to the current node and the distance from the current node to the destination node is smaller than the distance currently assigned to the destination node.
4. Repeat previous step for all possible destination nodes.
5. Designate the current node as finished.
6. Set the current node to the node that was already visited, wasn't finished and has the least distance. If such node does not exist, then the process is finished or else repeat steps 3-5.

More detailed information and examples are provided by Abiy et al. (2017).

The first step in any task that involves graphs is to create the graph itself. The simplest idea is to store list of nodes and list of edges. However, to make it easier to find edges from given node maps are usually used (Python Software Foundation 2017a). Furthermore, it possible to put the cost by using given edge by using tuples as it may be seen in Figure 14.

```
def createGraph():
    edges = {0: [(1,3), (2,4)],
             1: [(0,3), (3,5), (4,6)],
             2: [(0,4), (3,2), (6,7)],
             3: [(1,5), (2,2), (4,1), (7,22)],
             4: [(1,6), (3,1), (5,6)],
             5: [(4,6), (7,8)],
             6: [(2,7), (7,10)],
             7: [(3,22), (5,8), (6,10)]}
    graph = {'edges': edges}
    return graph
```

Figure 14. Creating a graph using map, lists and tuples in Python

In a such task an imperative programming is preferable than a functional one as changing states and conditionals are rather complicated tasks in functional programming as may be seen on Figure 15. For comparison Figure 16 shows same algorithm in Python which is more verbose but much simpler to write and understand. Also, OOP allows to create classes to represent nodes of a graph shown on Figure 17. Defining helper functions inside classes simplifies the work with graphs.

```
findShortestDistances::Map.Map Origin (Map.Map Destination Cost) -> Origin -> Map.Map Node Distance
findShortestDistances edges origin = calculateOneNode (Map.keys (edges Map.! origin)) (Map.singleton origin 0) origin (Set.singleton origin)
  where calculateOneNode needToVisit shortestDistances currentNode calculated
        | not $ null needToVisit = calculateOneNode (tail needToVisit) newShortestDistances currentNode calculated
        | not $ null possibleNextNodes = calculateOneNode newNeedToVisit shortestDistances nextNode (Set.insert nextNode calculated)
        | otherwise = shortestDistances
  where neighbor = head needToVisit
        newDistance = (edges Map.! currentNode) Map.! neighbor + shortestDistances Map.! currentNode
        newShortestDistances = if neighbor `Map.notMember` shortestDistances || newDistance < shortestDistances Map.! neighbor
                               then Map.insert neighbor newDistance shortestDistances
                               else shortestDistances
        possibleNextNodes = Set.toList $ Set.difference (Map.keysSet shortestDistances) calculated
        nextNode = fst $ foldr1 (\x y -> if snd x < snd y then x else y) $ map (\x -> (x, shortestDistances Map.! x)) possibleNextNodes
        newNeedToVisit = filter (`Set.notMember` calculated) (Map.keys (edges Map.! nextNode))
```

Figure 15. The Dijkstra algorithm in Haskell


```

def findShortestDistances(graph, index):
    edges = graph['edges']

    shortestPaths = {index:0}
    calculated = set()
    currentIndex = index

    while True:
        calculated.add(currentIndex)
        needToVisit = set()
        for edgePair in edges[currentIndex]:
            if edgePair[0] not in calculated:
                needToVisit.add(edgePair)

        for edgePair in needToVisit:
            if edgePair[0] not in shortestPaths or shortestPaths[currentIndex] + edgePair[1] < shortestPaths[edgePair[0]]:
                shortestPaths[edgePair[0]] = shortestPaths[currentIndex] + edgePair[1]

        possibleNextNodes = set(shortestPaths.keys()) - calculated

        if not possibleNextNodes:
            break
        else:
            currentIndex = reduce(lambda x, y : x if x[1] < y[1] else y,
                map(lambda x : (x, shortestPaths[x]), possibleNextNodes))[0]

    return shortestPaths

```

Figure 16. The Dijkstra algorithm in Python

```

public class Node {
    private Map<Node, Integer> neighbors = new HashMap<>();

    private int id;

    Node(int id) {
        this.id = id;
    }

    void addNeighbor(Node destination, int cost) {
        neighbors.put(destination, cost);
        destination.neighbors.put(this, cost);
    }

    Set<Node> getNeighbors() {
        return neighbors.keySet();
    }

    int getCostToNeighbor(Node destination) {
        return neighbors.get(destination);
    }

    @Override
    public String toString() {
        return "Node: " + id;
    }
}

```

Figure 17. *Node* class definition in Java, with *addNeighbour* creating a bidirectional edge

Table 9. Results of the Dijkstra test

	Time	Max memory
Haskell	< 0.001 s	42 KB
Python	0.0003 s	10.59 MB
Java	0.159 s	3.48 MB

The results are presented in Table 9. The time consumption in this test shows that Java wastes too much time working with classes where Haskell and Python which has great internal arrays. The memory footprint is may be considered meaningless as the only test with different value was the summing test on Java using *BigInteger*.

Implementation of Dijkstra algorithm on Haskell was the tough task which is doable but requires good understanding of recursion. In Java and Python everything was much simpler because of the cycles and mutability of variables. However, some subtasks were made in Java and Python using functional features.

5.5 Linear equations test

The best way to solve system of linear equation on a computer is to transform it into the matrix form. For example, there is given the following system of equations:

$$x + y + z = 6 \quad (202)$$

$$2y + 5z = -4$$

$$2x + 5y - z = 27$$

To transform this to the matrix form there should be the matrix with coefficients, the vector with variables and the vector with the right-hand side numbers.

Converting this system will yield the following matrix and vectors:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix} \quad (203)$$

The general form of the system of equation in the matrix form is following:

$$AX = B \quad (204)$$

There are three important facts about matrix multiplication shown in Equations 205-207. Where A^{-1} denote the inverse matrix of A and I is an identity matrix.

$$A^{-1}A = I \quad (205)$$

$$IA = A \quad (206)$$

$$\text{if } X = Y \text{ then } AX = AY \quad (207)$$

Using these two facts it is possible to make the following transformations:

$$AX = B \rightarrow A^{-1}AX = A^{-1}B \rightarrow IX = A^{-1}B \rightarrow X = A^{-1}B \quad (208)$$

One of the ways to find the inverse matrix is to find the adjugate of the matrix and divide it by determinant. The adjugate is defined as the matrix where each value x_{ij} is replaced by the corresponding minor M_{ji} . The minor M_{ij} may be found as determinant of the original matrix with row i and column j removed. The determinant may be found in a variety of ways where the most common ones are LU decomposition and Laplace expansion, where latter will be used. Definitions and examples may be found at MathWorld (2017).

In the end, solving a system of linear equations becomes a problem of finding the adjugate with minors, multiplying it by a vector and dividing it by the determinant. Corresponding snippets of code may be seen on Figures 18-22.

```
getAdjugate :: (Fractional a) => [[a]] -> [[a]]
getAdjugate matrix = map (map (\tuple -> uncurry getMinor tuple matrix)) matrixOfIndices
  where len = length matrix
        matrixOfIndices = map (\y -> map (\x ->(x,y)) [0..(len - 1)]) [0..(len - 1)]
```

Figure 18. The function to get the adjugate in Haskell

```

public static double getDeterminant(List<List<Double>> matrix) {
    if(matrix.size() == 2) {
        double a = matrix.get(0).get(0);
        double b = matrix.get(0).get(1);
        double c = matrix.get(1).get(0);
        double d = matrix.get(1).get(1);

        return a * d - b * c;
    } else {
        double sum = 0;

        for(int y = 0; y < matrix.size(); y++) {
            sum += matrix.get(0).get(y) * getMinor(0, y, matrix);
        }

        return sum;
    }
}

```

Figure 19. The function to find the determinant in Java using Laplace Expansion

```

def getDeterminant(matrix):
    if len(matrix) == 2:
        a = matrix[0][0]
        b = matrix[0][1]
        c = matrix[1][0]
        d = matrix[1][1]
        return a * d - b * c
    else:
        return sum(map(lambda y : matrix[0][y] * getMinor(0, y, matrix), range(len(matrix))))

```

Figure 20. The function to find the determinant in Python using Laplace expansion

```

multiplyMatrixAndVector :: (Fractional a) => [[a]] -> [a] -> [a]
multiplyMatrixAndVector matrix vector = map (sum . zipWith (*) vector) matrix

```

Figure 21. The function to multiply a matrix and a vector in Haskell

Little support function *uncurry* has signature $(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$ and helps when function takes two arguments, but we have packed them into tuple. As unfolding tuple usually requires assigning variables or usage of function like *fst* this function helps to tide a code by passing to it tuple directly to a function.

```

public static double getMinor(int i, int j, List<List<Double>> matrix) {
    List<List<Double>> minorMatrix = new ArrayList<>();

    for(List<Double> row : matrix) {
        minorMatrix.add(new ArrayList<Double>(row));
    }

    minorMatrix.remove(i);

    for(List<Double> row : minorMatrix) {
        row.remove(j);
    }

    return Math.pow(-1, i + j) * getDeterminant(minorMatrix);
}

```

Figure 22. The function to find a minor in Java

The results of the test are presented in Table 10. The time consumption predictably shows lightning-like speed of Haskell which again cannot be measured by internal tools. Python is surprisingly much faster than Java. The most probable bottleneck in Java is *List* which is extremely slow compared to internal arrays in Python and Haskell.

Table 10. Results of the linear equations test

	Time	Max memory
Haskell	< 0.001 s	42 KB
Python	0.00076 s	10.637 MB
Java	0.16 s	9.87 MB

The implementation of this task had different challenges on different languages. For example, Haskell's lack of cycles forced to create the matrix with indices and then to map from these indices to minors to get the adjugate as may be seen on Figure 18. In other languages two nested cycles were used which is more intuitive and easier to understand.

The functional approach was used in Java and Python languages when it was preferable. In Python, for example, the weighted sum of minors to find the determinant using Laplace Expansion is using mapping, though it is possible to

implement it like in Java using cycle. The amount of lines required to use these two different approaches may be compared on Figures 19 and 20.

Division of matrix by scalar is done with mapping which is intuitive. But, functional features in Java are just like whole code in Java in general, are too verbose and may look less appealing than in other languages. First, *Collection* (*Set*, *List*, *Map*) should be converted to *Stream* using the same-named function. Functional operations in Java can be performed only with *Stream*. Afterwards, *Stream* should be either collected back into some *Collection* or reduced (folded) to a single value. On Figures 23-25 it is easy to notice how a short mapping operation in Haskell grows in Python and Java.

```
map (/determinant) multiplied
```

Figure 23. Division of vector by scalar in Haskell

```
answer = list(map(lambda x : x / determinant, multiplied))
```

Figure 24. Division of vector by scalar in Python

```
List<Double> answer = multiplied.stream().map(x -> x / determinant).collect(Collectors.toList());
```

Figure 25. Division of vector by scalar in Java

6 CONCLUSIONS

Haskell is an amazing choice for writing applications that define results and not process it. Usually it is said that Haskell suits any mathematical task, but the Dijkstra test showed that though it is still mathematics, it was a rather complicated task. In the derivatives and linear equations tests Haskell outperformed Java and Python completely because derivatives are just defined which is the best possible case for Haskell. All tests showed the low memory footprint and lowest time consumption which indicates the high efficiency of optimized Haskell code.

Overall, functional programming is a powerful tool that simplifies specific tasks, but sticking to purely functional programming is over-limiting and makes some things too complicated. Some compromise should be achieved to utilize

functional features but not to lose typical imperative programming ones. Python, Java or even C++ all have functional features added to them, which means that functional approach is popular. But, as it was seen Java add too much verbosity for the simple functional features like mapping. In some of languages functional features may look ugly, and thus, shouldn't be used even if they are added there. Instead, mixed imperative-functional languages should be used in tasks that require the functional features sometimes. The modern examples of such languages include F# and Scala. These languages were aimed to have the functional features without forfeiting everything from the imperative paradigm in the favor to the functional one.

REFERENCES

Abiy, T., Pang, H., Khim, J., Ross, E., Williams, C. 2017. Dijkstra's Shortest Path Algorithm. WWW document. Available at: <https://brilliant.org/wiki/dijkstras-short-path-finder/> [Accessed 21 November 2017]

Aquilina, M. 2017. Python Tools package. WWW document. Available at: <https://atom.io/packages/python-tools> [Accessed 21 November 2017]

Atom-Haskell. 2017. Atom-Haskell official website. WWW document. Available at: <https://atom-haskell.github.io/> [Accessed 20 October 2017]

Boyer, S. 2014. Super quick intro to monads. WWW document. Available at <https://www.stephanboyer.com/post/83/super-quick-intro-to-monads> [Accessed 16 October 2017]

Braithwaite, R. 2017. Why Recursive Data Structures? WWW document. Available at: <http://raganwald.com/2016/12/27/recursive-data-structures.html> [Accessed 13 October 2017]

Cygwin. 2017. Cygwin official website. WWW document. Available at: <http://www.cygwin.com/> [Accessed 21 November 2017]

GitHub Inc. 2017. Atom official website. WWW document. Available at: <https://atom.io/> [Accessed 20 October 2017]

Haskell. 2011. Pointfree. WWW document. Available at: <https://wiki.haskell.org/Pointfree> [Accessed 29 October 2017]

Haskell. 2017a. Haskell official website. WWW document. Available at: <https://www.haskell.org/> [Accessed 20 October 2017]

Haskell. 2017b. Haskell Performance Resource. WWW document. Available at: <https://wiki.haskell.org/Performance/GHC> [Accessed 21 November 2017]

Hudak, P. 2008. A Brief and Informal Introduction to the Lambda Calculus. PDF document. Available at: <http://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf> [Accessed 11 October 2017]

Jung, A. 2004. A short introduction to the Lambda Calculus. PDF document. Available at: <http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf> [Accessed 10 October 2017]

Lipovača, M. 2011. Learn You a Haskell for Great Good!. WWW document. Available at: <http://learnyouahaskell.com/> [Accessed 10 October 2017]

MathWorld. 2017. Mathematics online encyclopedia. WWW document. Available at: <http://mathworld.wolfram.com/> [Accessed 21 November 2017]

McCarthy, J. 2012. Church Encoding. Blog. Available at: <http://jeapostrophe.github.io/2012-08-20-church-e-post.html> [Accessed 13 November 2017]

Michaelson, G. 2011. An Introduction to Functional Programming Through Lambda Calculus. Mineola, New York: Dover Publications, Inc.

Nadeau, D. R. 2008. Java tip: How to get CPU, system, and user time for benchmarking. WWW document. Available at: http://nadeausoftware.com/articles/2008/03/java_tip_how_get_cpu_and_user_time_benchmarking [Accessed 20 October 2017]

Oracle. 2014. What's New in JDK 8. WWW document. Available at: <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html> [Accessed 20 November 2017]

Oracle. 2016. VisualVM official website. WWW document. Available at: <https://visualvm.github.io/> [Accessed 21 November 2017]

Oracle. 2017. Java official website. WWW document. Available at: <http://www.oracle.com/technetwork/java/index.html> [Accessed 20 October 2017]

Prokopev, V. 2017. The practical part source code repository. WWW document. Available at: <https://github.com/vadimjprokopev/Thesis> [Accessed 21 November 2017]

PyPA. 2016. *pip* documentation. WWW document. Available at: <https://pip.pypa.io/en/stable/> [Accessed 21 November 2017]

Python Software Foundation. 2017a. Implementing graph. WWW document. Available at: <https://www.python.org/doc/essays/graphs/> [Accessed 21 November 2017]

Python Software Foundation. 2017b. *jedi* documentation. WWW document. Available at: <https://pypi.python.org/pypi/jedi> [Accessed 21 November 2017]

Python Software Foundation. 2017c. *psutil* entry page. WWW document. Available at: <https://pypi.python.org/pypi/psutil> [Accessed 20 October 2017]

Python Software Foundation. 2017d. Python official website. WWW document. <https://www.python.org/> [Accessed 20 October 2017]

Reddy, U. 2009. Lambda Calculus Examples. PDF document. Available at: <http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/lambda-calculus-handout.pdf> [Accessed 10 October 2017]

Riedel, H. V. 2016. Haskell Prime 2020 committee has formed. WWW document. Available at: <https://mail.haskell.org/pipermail/haskell-prime/2016-April/004050.html> [Accessed 20 November 2017]

Rowland, T. 2017. Church-Turing Thesis. WWW document. Available at: <http://mathworld.wolfram.com/Church-TuringThesis.html> [Accessed 11 October 2017]

Sadovnychiy, D. 2017. Python Autocomplete Package. WWW document. Available at: <https://atom.io/packages/autocomplete-python> [Accessed 21 November 2017]

The Eclipse Foundation. 2017. Eclipse official website. WWW document. Available at: <https://www.eclipse.org/> [Accessed 21 November 2017]

The PyPy project. 2017. How fast is PyPy? WWW document. Available at: <http://speed.pypy.org/> [Accessed 27 October 2017]

Turner, D. A. 2012. Some History of Functional Programming Languages. PDF Document. Available at: <https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf> [Accessed 20 November 2017]