FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Comparison of Haskell and F# Programming Languages for Enterprise Applications Development
Student:	Nasiha Maleškić
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Information Systems and Management
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

The goal of the thesis is to compare two significant functional programming languages: F# and Haskell with the respect to enterprise applications development.
1. Acquaint yourself with principles of pure functional programming paradigm, F# and Haskell (and possibly similar languages) in the context of enterprise applications.
2. Select a suitable case study of a project realised in F# and perform a comparison with a hypothetical Haskell implementation:
   a. Comment on the languages themselves.
   b. Analyse the ecosystems of the languages.
   c. Analyse other aspects such as tools support, community, documentation, etc.
3. Formulate conclusions of your analysis, especially strong and weak points of Haskell as opposed to F#.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 22, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Comparison of Haskell and F# Programming Languages for Enterprise Applications Development

Nasiha Maleškić

Department of Software Engineering
Supervisor: doc. Ing. Robert Pergl, Ph.D.

May 15, 2019

Acknowledgements

I want to thank my supervisor doc. Ing. Robert Pergl, Ph.D. for his guidance and valuable advice. I express the deepest gratitude to my family for the constant encouragement and support I never cease to receive from them. I also thank my partner for his love and support. I want to thank Filippo Ghibellini for his help regarding Haskell and thank Roman Provaznik for sharing his knowledge of F#.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Maleškić, Nasiha. *Comparison of Haskell and F# Programming Languages for Enterprise Applications Development.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce porovnává dva funkcionální jazyky Haskell a F#. Oba jazyky jsou analyzovány na základě kritérií, které by byly použity při výběru nejvhodnějšího jazyka pro vývoj podnikové aplikace. Práce také vysvětluje event sourcing a jeho implementaci v obou jazycích. Důvodem, proč je Haskell oproti F# lepší pro event sourcing je, že je referenčně transparentní. Dále práce vysvětluje, že je jednodušší psát frontend aplikace v F# ve srovnání s Haskellem, z dúvodu toho, že F# není čistě funkcionální. V závěru se v práci dospěje k tomu, že ve většině případů je F# vhodnější pro vývoj podnikových aplikací.

Klíčová slova F#, Haskell, vývoj podnikových aplikací, porovnání funkcionálních jazyků

Abstract

This thesis compares two functional languages Haskell and F#. Both languages are analyzed based on the criteria that would be used in order to choose the most suitable language for developing an enterprise application. It explains event sourcing and its implementation in both of the languages and why Haskell is better for event sourcing due to the fact it is referentially

transparent. It explains how it is easier to write F# frontend applications compared to Haskell, due to its impurity. It concludes that it seems, that in most of the cases, F# is more suitable for enterprise application development.

Keywords F#, Haskell, enterprise applications development, functional languages comparison

Contents

List of Figures

Introduction

State of enterprise applications development is such that the majority of development is in object-oriented languages. Projects written in functional languages pique the question of whether this could be a more efficient way for development. With functional languages on the rise, and after working for six months as an F# developer, I was motivated to research more about them.

Research is focused on functional languages and their specific characteristics. The analysis is focused on comparing two significant functional languages: F# and Haskell. I analyze whether using functional languages (and more specifically which ones) in enterprise applications development is beneficial or the contrary. In the last chapter, I take a look at the case study project written in F# from a real company. I compare the F# approaches to hypothetical Haskell implementation and comment on the difference. This further helps us gain better insight.

I specifically compare Haskell and F#. Haskell is chosen due to the fact it is the most popular purely functional language. F# was chosen, not only for personal reasons but also because it is one of the biggest functional languages on the rise. It is also one of the best-payed technologies in the last few years.

Deciding on a language to use in enterprise applications development can be difficult. Choosing the right language for your needs can be extremely beneficial in terms of time spent in development, software bug reduction, developer's motivation and client satisfaction.

This bachelor thesis intends to present the advantages and disadvantages of one language and the other, as well as demonstrate their differences and their suitable applications. This can help developers make a more informed decision during the process of choosing the programming language for their needs.

Chapter 1

Goals and approach

The goal of this thesis is to compare two significant functional programming languages: F# and Haskell with respect to enterprise applications development.

I perform a review of the functional programming paradigm, as well as some of the languages that have influenced Haskell and some of those that were influenced by Haskell. Furthermore, I review Haskell and F#. I analyze the development tools from both Haskell and F#, as well as what it is like to use those languages in an enterprise environment. I finish the Analysis chapter with a comparison of Haskell and F# in regards to specific elements that can be crucial when choosing a language for enterprise applications development. In the final chapter, I take a look at a real-world application built in F# and its implementation, comment on it and compare it with a hypothetical Haskell implementation. The focus is on desktop applications, whereas frontend was chosen due to the fact that it provides a nice contrast to event sourcing.

Chapter 2

Review

2.1 Functional languages

In the following paragraphs, I will explain basic ideas behind the functional programming paradigm as well as introduce characteristics that are common for all functional languages. Features we will focus on are those that may be of the greatest benefit for the enterprise applications development.

2.1.1 Introduction

Functional programming paradigm is called that way because the programs consist entirely of functions. These are called first-class functions since they can be treated just like any other value (integer or string for example). They can be passed as arguments or returned as a result of a function. The distinguishing characteristic of the functional programming paradigm lies in focusing on what needs to be done, rather than on how it needs to be done. Functional languages are usually divided into pure and impure ones. Generally accepted definition is that those functional languages that allow for side effects are impure.

Haskell is a purely functional language, meaning it does not allow any side effects at all. There are still discussions as to what is considered a pure functional language and what impure. For our purposes, we need not dive into those discussions, as we will focus on the elements of functional programming that are beneficial for enterprise applications development. F# is a multi-paradigm language, allowing for imperative and OOP elements as well.

Common features of all functional languages: immutable data, referential transparency, higher-order functions and lazy evaluation.

2.1.2 Immutable data

Instead of altering the original values, copied values are altered. Purely functional programs typically operate on immutable data.[1] When modifying an immutable data structure, we expect that the old and the modified version will be available for further processing. We take the old data structure, copy it and modify the new version. Data structures that allow multiple versions are called persistent.[2]

2.1.3 Referential transparency

An expression always evaluates to the same result. *We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs.*[3]

2.1.4 Lazy evaluation

Lazy evaluation means that functions are evaluated only when needed. Lazy evaluation is also called call-by-need evaluation because the evaluation of an expression is done only when needed as opposed to eager evaluation (also called call-by-value) where the value of an expression is evaluated immediately as it is assigned to the variable.[4] The latter is more widespread, but both of them have their advantages as well as disadvantages. Laziness can be less efficient and less predictable, the latter one being true even for experienced programmers.[5]

2.1.5 Higher-order functions

Higher-order functions are functions that take other functions as their arguments. Some of the most common higher-order functions in functional languages (some of them are implemented even in non-functional languages) are: *reduce*, *filter*, *find* and *map*. An example of a higher order function in F# is List.map. List.map in F# is used to transform a list into a new list by applying the function onto each element of the list.[6]

```
let increaseByOne x = x |> List.map (fun x -> x + 1)
printf "%A" ([1; 2; 3; 4] |> increaseByOne)
```

Listing 1: List.map example

2.2 Examples of languages that influenced Haskell

I have chosen to research and describe the following languages because according to the developers they are the ones that have had the most influence on Haskell development.[5] From these languages, we can also see what some of the features that the committee working on Haskell decided to include as they were, those that they changed and those that they completely left out. It also gives an insight into the history of how Haskell came to be.

2.2.1 Miranda

Miranda is a non-strict, purely functional programming language with polymorphic types. *Non-strict semantics allows your language only to evaluate the things it needs to.*[1] Non-strict semantics can hurt performance because the program needs to decide in advance what needs to be evaluated and what does not. The advantage it gives us is cleaner code, one example of this being that unnecessary arguments will not be evaluated.

It supports pattern matching, currying, list comprehension, polymorphic strong typing, user-defined types and much more.

Miranda is a strongly typed language. There are three mechanisms for user-defined types: type synonyms, algebraic types and abstract types. Type synonyms permit the user to define a name for an already existing type. The symbol == is used. Algebraic data types are used when introducing a new concrete data type. The symbol ::= is used.[7]

2.2.1.1 Miranda and Haskell

Back in 1987, when it came to deciding whether to write a new programming language completely from scratch or to take another one as the basis for implementation - all eyes were on Miranda. It was developed by David Turner's company called Research Software and as such was a commercial language. Pure, well-designed with robust implementation and lazy evaluation it seemed like the perfect starting point for this new language that was later to be named Haskell. David Turner had declined, since this new language was to be used differently than Miranda, the main difference being that Haskell would not be commercial. *Nevertheless, Haskell owes a considerable debt to Miranda, both for general inspiration and specific language elements that we freely adopted where they fitted into our emerging design.*[5]

Haskell was aspiring to be many things that Miranda already was. Therefore there are numerous similarities between the two languages. Haskell gained more popularity over Miranda for several reasons. Miranda was not a part of the public domain, and it remained a commercial language, whereas Haskell was available for free for everyone. Miranda ran only on Unix, whereas Haskell

ran on Windows as well. Alongside the new ideas, Haskell implementations were also rapidly improving which is why it eventually displaced Miranda.[5]

2.2.2 ML

ML stands for Meta Language, which is a term used for a form of a language or a set of terms used for describing and analyzing other languages.[8]

When talking about ML, it is essential to mention the two distinct phases of program execution. First one is the static phase, determining whether the program is well-formed before it is run.[8] The second phase is the dynamic phase when the program is run. In Standard ML there is also a third phase, which comes before the two. It is called parsing, and *it determines the grammatical form of a declaration.*[9] Another crucial characteristic is the language structure. It consists of three levels. The lowest level is called Core, which provides many phrase classes. The middle-level concerns with programming larger applications, called Modules. The very small upper level is called Programs.[8]

Its two dialects are Standard ML and Caml (Categorical abstract machine language) with OCaml as its main implementation.

2.2.2.1 Standard ML

Standard ML is a type-safe programming language. It encourages functional paradigm, but imperative style can be used where needed. Therefore it is not considered a pure functional language. It is a statically typed language and as such does the verifying and type constraints enforcing at compile time. It has an extensible type system and supports polymorphic type inference.[10]

2.3 Examples of languages influenced by Haskell

I have chosen to research and write about Agda and Idris primarily for their research value. They are an insight into what the languages of the future might look like. Idris very young, but it is built on top of Haskell, and it provides us with features that can be considered improvements over the greatly more popular Haskell.

2.3.1 Agda

Agda is a dependently typed programming language and proof assistant.[11]

Dependent type is a type whose definition depends on a value. In dependently typed languages, the distinction between values and types is not always clear. *In Hindley-Milner style languages, such as Haskell and ML, there is a clear separation between types and values. In a dependently typed language,*

the line is more blurry types can contain (depend on) arbitrary values and appear as arguments and results of ordinary functions.[12]

Hindley-Milner is a type system named after the logician J. Roger Hindley, who first described it and Robin Milner, a computer scientist who later rediscovered it. It is a type system that has an algorithm for inferring types without them being previously declared.

Function definitions must cover all possible cases, and they have to be terminating. Naming in Agda is very liberal, all nonwhite Unicode characters are allowed, few exceptions being parentheses and curly braces.[12]

2.3.2 Idris

Idris is a general purpose pure functional programming language with dependent types.[13]Like Agda, it can be used for writing programs as well as proofs. The main difference between the two is the fact that Idris was developed for general purpose programming from the early start, whereas Agda started as a tool for proving theorems. Even so, at the moment it is primarily used for research, and according to the developers, it is yet to be made into a system that could be used in production.[13]

2.3.2.1 Eager evaluation

The main difference between Haskell and Idris is the fact that Idris has eager evaluation. Idris does encompass opt-in laziness. This can be seen as an advantage over Haskell since laziness can lead to problems with performance and it can make it *difficult to reason about the predictability of a program.*[14]

One of the many other differences, which some may consider as improvements, is the fact that Idris strings are not lists. Haskell string is a list of characters, but in Idris String is a built-in.

2.4 Haskell

Haskell is a general-purpose, purely functional language. Haskell is statically typed and has a type system that encompasses type inference. Its main implementation is Glasgow Haskell Compiler.[1]

2.4.1 Syntax

Haskell wanted to distinguish itself from the imperative languages even in the syntax of the language. Instead of using braces, Haskell uses indentation for delimiting blocks, and instead of semicolons, a newline is used as a separator. The developers have left the ability to override the layout rules in order to give more freedom to the users. Haskell has 21 keywords that cannot be used as variable names.[5]

2.4.2 Haskell in industry

In the following paragraphs, I will give examples of three projects within big companies that were done in Haskell.

2.4.2.1 Fighting spam at Facebook

Sigma is a system at Facebook used for fighting against malicious actions on the famous social network. For every action, whether it is a post, a like or a message sent - the system checks the policies regarding that very action to prevent spam, phishing and other similar actions before they can harm the users.

At first, the system was written in in-house FXL language, but then they started looking for something better. On the list of their requirements was that the language was to be purely functional and strongly typed. This was needed in order for policies not to inadvertently affect each other and so that they are easier to test in isolation.

Implicit concurrency employment should be easy, so that *engineers writing policies can concentrate on fighting spam and not worry about concurrency.* All existing concurrency implementations in Haskell are explicit, so they looked for different options. Automatic batching and concurrency were to be addressed by The Haxl framework. The Haxl framework is a Haskell library that *simplifies access to remote data*; *it can batch multiple requests to the same data source and request data from multiple data sources concurrently.*

Developer team working at Facebook's Sigma project wanted to create a tool that had a fast performance, was efficiently handling the requests and correctly applying the rules. On top of that, they wanted to have clean, easily readable code. That is the short story of how they have come to choose Haskell.[15]

2.4.2.2 Googles tool for clusters of virtual machines

Ganeti is a management tool that was developed under Google's IT system administration group. Before adding Haskell to it, the entire project was written in Python. Their goal was to develop tools that can automate the layout computation in order to best use the resources of each physical machine. Since actual policies for the layout of a virtual machine can differ according to different site policies, they decided to leave that to the external scripts and that the Ganeti itself should just implement the mechanism. They have decided to write a so-called cluster balancer, that takes the current state of the cluster and decides on the next best state.

There are two main reasons why the developers have decided to write this component in Haskell. All of the problems they are trying to solve are numerical, and those kind of problems are easy to model in a pure domain.

Another thing that the developers have praised is how short the code base is. The learning curve for Haskell is a rather steep one, but once you master it, you can write short code that does a lot.[16]

2.4.2.3 IMVU goes from PHP to Haskell

At IMVU, an avatar-based social platform, they have made a switch in 2013 from PHP to Haskell to build several of the REST APIs. At the beginning of the company, they have used PHP as the application server language. Eventually, the client base grew a lot bigger, and the company was looking for alternatives in order to improve the performance. They started experimenting with Haskell. They have faced the same concerns many have when choosing not to use Haskell for their projects. Training developers is one of them, Haskell being infamous for its steep learning curve. They say that training someone to be productive in Haskell was no harder than in PHP, especially if they have already had some prior functional programming experience. They have praised ease of deployment and refactoring.

2.5 F#

F# is a strongly typed, first-class .NET programming language designed by Don Syme and others at Microsoft Research.[17] It is a functional-first language, meaning that it encourages the use of functional paradigm but allows for imperative and OOP methods as well.

F# is interoperable with C#, and it natively supports all of the .NET classes, interfaces, and structures. That is one of the reasons why it can be an excellent way for companies to start adopting or transitioning to functional paradigm slowly. You can use the existing code in C# and on top of it start adding F# code.[18]

2.5.1 Syntax

F# syntax is simple and straightforward. Indentation is used for delimiting blocks of code. There are no needless brackets. Operators greatly improve readability, the three most important ones and the ones used most frequently being: pipe forward operator, pipe backward operator and composition operator. With proper variable naming and proper operator usage - it can often be quite clear what the code does just from looking at it. We read the code from left to right, just as we are used to doing from the natural languages. This can be very beneficial when the developers are trying to explain their code to someone who has no experience programming.

Chapter 3

Analysis

3.1 Haskell development tools analysis

3.1.1 Compiler

Glasgow Haskell Compiler (GHC) is an interactive, open-source compiler for Haskell. It is written in Haskell, but the runtime system is implemented in C and C–.[1] It consists of two main components: the batch compiler and the interactive environment called GHCi. With GHCi you can interactively evaluate expressions and interpret the program.[19]

GHC provides support for parallel and concurrent programming. As a way for handling concurrent threads, an abstraction called Software Transactional Memory (STM) was added to GHC 6.4. *The main benefits of STM are composability and modularity.*[1] In order to use STM, the programmer needs to guarantee atomicity and isolation.[20] This is ensured by the atomically combinator. Once the block of actions of transactions is entered it cannot be affected by other threads nor can we see the modifications made until we exit. Upon exiting, if no one else was modifying the same data, our changes will be immediately visible. Otherwise, our actions will be discarded.[21]

3.1.2 Testing

Testing is an integral part of the software development life cycle. Testing is carried out in order to detect any software defects in order to correct and remove them before deployment.[22] Haskell has *the expressive type-system, which allows for complex invariants to be enforced statically making it impossible to write code violating chosen constraints.*[23] Apart from this, two main testing mechanisms are the standard unit testing (via the HUnit library) and the more sophisticated type-based property testing (via the QuickCheck framework).[23] It is essential to mention HSpec, a testing framework which integrates with both QuickCheck and HUnit, as well as SmallCheck.

3.1.2.1 Property based testing with QuickCheck

In property-based testing, we tell the program how the output should look like based on the input. The program then generates a large number of cases to test that. When writing property-based tests, we have to think carefully about the specifications. What kind of input is supported and what kind of output is expected. With property-based testing, we can cover a more substantial amount of tests, compared to writing them by hand. We are also able to discover problems with subtle corner cases we could have otherwise forgotten.[23] QuickCheck is a Haskell library that generates test cases (according to the given specifications) to test program properties.[24]

Following is an example of Bubble Sort algorithm written in Haskell. Ord is a typeclass for types that have an ordering. x:xs represents a list where x is the first element (head), and xs is the rest of the list (tail). x:y:xs is a pattern that says that this is a list with at least two elements x and y. The remaining sublist is xs, and it may be empty.

```
module Main where

bubbleSort' :: Ord a =>  [a] -> [a]
bubbleSort' (x:y:xs) =
 if x > y then y:bubbleSort' (x:xs)
 else x:bubbleSort' (y:xs)
bubbleSort' xs = xs

bubbleSort :: Ord a => [a] -> [a]
bubbleSort [] = []
bubbleSort lst =
 let t1 = bubbleSort' lst
 in bubbleSort (init t1) ++ [last t1]

main :: IO ()
main = do
putStrLn $ show $ bubbleSort [1,3,5,2,4,0]
```

Listing 2: Bubble Sort in Haskell[25]

To demonstrate property-based testing in Haskell we will use previously mentioned QuickCheck library. We need to import it first.

```
import Test.QuickCheck
```

When writing property-based tests, we need to think about the properties of the function that we want to test. We are testing a sort algorithm. Therefore the first and the most obvious property is that the algorithm needs to sort numbers properly. We will test against the standard sort library. This is known as testing against the model implementation. We know that the standard sort library is a model that works and therefore we can confidently test our function against it.

When writing QuickCheck tests, it is a convention to start them with prop_ to distinguish them from the rest of the code.

```
prop_sort_model xs = sort xs == bubbleSort xs
```

3.1.2.2 Haskell Program Coverage

HPC (Haskell Program Coverage) is an extension to the compiler that can tell us the exact percentage of the code that was executed when the tests are run. It can also tell us precisely which parts of the program were executed. It comes with a utility that can generate visual coverage display; for example, it can create an HTML file with pretty graphs that make it simple to see which functions were left untested. It is useful because that way we know exactly which tests to add.[23]

3.1.3 Debugging

A read-eval-print loop (REPL) is a simple environment that takes a single expression, evaluates it and returns the result. In the case of Haskell REPL is GHCi, the GHCs the interactive environment. If separate functions are tested in REPL before being added to the modules, the developer can catch a lot of bugs here. GHCi also contains a debugger that can be used to find bugs that are more difficult to see. In this debugger, you can set breakpoints and stop the computation and examine the value of variables.[26] Breakpoints can be nested so that one breakpoint can trigger the second one. It also enables for the execution to be *single-stepped*, which would be like setting a breakpoint at every point in the program.[26] Debug.trace are functions for so-called "printf debugging" where we choose to output information at critical points in the program which help to indicate us where the problem is. *The trace function outputs the trace message given as its first argument, before returning the second argument as its result.*[27]

3.1.4 Documentation

Haddock is a tool for automatically generating documentation from Haskell source code. Despite being primarily developed for Haskell libraries, it can be

helpful when documenting any code. It supports several formats with HTML and partial support for generating LaTeX.[28]

3.2 Using Haskell in a company

Before deciding on using Haskell in a company or for a project one needs to consider a wide array of criteria. Sorting a list of criteria by the importance is an impossible task since they are all relative to each other. Projects application domain could be perfectly suitable for Haskell, but if you do not have the developers with the necessary expertise, it can not help. You may have a team of expert Haskell developers, but using Haskell to build a mobile application would be pointless considering the alternatives.

The majority of criteria applies to both new companies and projects as well as the already established ones. In case of an established company, one has to consider the current company structure and its employee's skill set, the existing languages used at the company and how productive the transfer from the current technology to Haskell would be. Those participating in the decision process need to calculate the total cost of using new tools, educating the employees or hiring new ones, rewriting the code and the impact it will have on the company culture.

3.2.1 Community

Haskell is 40th on the TIOBE Programming Community index for September 2018 (indicates how popular a programming language is) list with 0.218% ratings. Top 18 programming languages on the list all have a rating above 1%.[29] It shows us that Haskell is not a prevalent language overall, but in the following paragraphs, I will give a list of examples of a growing Haskell community and its increasing potential for popularity.

Haskell is repeatedly on StackOverflow Developer Survey lists as one of the most loved and the most wanted technologies on the global scale. Haskell community is large and highly active on several mediums, such as mailing lists, IRC and StackOverflow.[30] There is even a special group dedicated to companies and everyone else who is either already using or looking to use Haskell commercially. The commercial Haskell group is meant for those willing to collaborate, improve the quality of tools and libraries to meet commercial standards and acquaint the greater public about the tools and discoveries through better documentation and tutorials.[31] FP Complete is an engineering company that will help you with commercial development using Haskell. They do so by offering you their expertise and consulting. They can help you build your product either partially or entirely. Some of the services they offer are providing the educational materials, tutoring your employees, writing the code and delivering the product.[30]

3.2.2 Human resources

One of the most important factors is whether actual people are willing to work with the given language or technology. Taking into account the companys flexibility we could look at not only local but also global talent pool. Haskell is generally criticized for having a steep learning curve.[32] Steep enough, that one would probably take at least a year to be productive in it and a few years to become an expert.

Considering the steep learning curve, one needs to expect higher salaries. According to the StackOverflow Developer Survey 2018, Haskell came out as the 13th top paying technology on a global scale.[33]

In Prague, all introductory programming courses are taught with object-oriented programming. None of them even have functional programming classes obligatory with one exception (Charles University - Faculty of Mathematics and Physics).

3.2.3 SWOT Analysis

3.2.3.1 Strengths

Testing Due to the fact that Haskell has an expressive static type system it allows for complicated invariants to be enforced statically. This makes it impossible to write code that would violate chosen constraints. That leads to having to test for fewer things than in other languages. Purity and polymorphism encourage writing modular, refactorable code that is easy to test. On top of it all, we also have previously mentioned framework QuickCheck and HUnit library that helps us build precise programs.[23]

Speed and parallelism *GHC implements some major extensions to Haskell to support concurrent and parallel programming.*[19]

Refactoring Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.[34] By refactoring the code, we can produce clean code that is easier to read and maintain. Code smells are any parts of the program that may indicate more significant problems. Code smells can be subjective. There are various norms and standards for different programming languages and styles as to what a code smell is. Redundant boolean check, unnecessarily long lines of code and duplicate code are examples of code smells in Haskell.[35]

HaRe is a refactoring tool for automated refactoring of Haskell code. It is still in development, but it currently supports several features, such as renaming, duplicating definitions and converting an if statement to a case statement.[36]

Following snippets demonstrate code before and after removal of a redundant Boolean check, which can be achieved with HaRe.

Before refactoring:

```
funcBoolReturn :: Bool -> Bool
funcBoolReturn p = p == True
```

Listing 3: Before refactoring

After refactoring:

```
funcBoolReturn :: Bool -> Bool
funcBoolReturn p =  p
```

Listing 4: After refactoring

3.2.3.2 Weaknesses

Learning curve Haskell has a steep learning curve, and it takes a long time for the developer to feel capable in the language.

Hiring Haskell wards developers off with its steep learning curve. It is considered a relatively obscure language which is another reason that developers are not tempted to learn it. Haskell is generally not taught in schools. These are some of the causes that lead to a significantly smaller pool of potential hires, especially when compared to already well-established languages such as Java. That is why finding the right people can be a challenge. Even after you find them hiring is a challenge as well, since salaries are more competitive than in mainstream languages.

3.2.3.3 Opportunities

Large companies Large companies such as Facebook and Google are doing projects in Haskell. Clients are more likely to accept a particular (even a novel) technology for their project if they are given a proof that similar (or at least big and stable) projects have already been done with that very technology.

3.2.3.4 Threats

Already popular languages Haskell may never become mainstream. No one can see the future, but we can look at the past. Languages that have worked in the past still work for us, there is a more significant

and more substantial code base, more tutorials, more support and more experience with those languages than with the relatively new Haskell. That is why many feel reluctant using a new language that may not even stay for good.

Multi-paradigm languages One of the threats to Haskell is programming languages that encompass several programming paradigms (including functional) such as F#. Developers are given more freedom to decide for themselves whether they feel more comfortable or confident to write a particular piece of code in imperative, object-oriented or functional style. It leads to more possibilities and therefore bigger interest.

Haskell reputation Haskell comes from an academic environment, and it is generally deemed as an academic and a challenging language to learn with its abstract, mathematical notations. Reputation takes time to build and also takes time to change. Haskellers, in particular, are not even very keen on improving the reputation, and thats what makes it a threat. If the Haskell community wants Haskell to become more mainstream, they will have to change their approach.

3.3 F# development tools analysis

One of the disadvantages is convincing the clients that a big project can be written in a functional language, which is still relatively unheard of, at least in mainstream circles. That is why when compared to other functional languages its interoperability with C# and native support of .NET come as an advantage.

3.3.1 Testing

Following is an example of Bubble Sort algorithm written in F#.

```
let bubbleSort (lst : list<int>) =
    let rec sort accum rev lst =
        match lst, rev with
        | [], true -> accum |> List.rev
        | [], false -> accum |> List.rev |> sort [] true
        | x::y::tail,
            _ when x > y -> sort (y::accum) false (x::tail)
        | head::tail, _ -> sort (head::accum) rev tail
    sort [] true lst
```

Listing 5: Bubble Sort in F#[1]

Similar to QuickCheck, FsCheck is a tool for testing F# (.NET) programs automatically. We will test the following properties that should be true for any sorting algorithm. Sorting the same list twice should always yield the same sorted list.

```
let ``sort should be same as sort`` sortFn aList =
  let result1 = aList |> sortFn
  let result2 = aList |> sortFn
  result1 = result2

Check.Quick (``sort should be same as sort`` bubbleSort)
```

Listing 6: FsCheck: Sorting the same list should always return the same result

Applying the sort function twice should yield the same result as applying the sort function once.

```
let ``sort once should be same as sort twice`` sortFn aList =
    let result1 = aList |> sortFn
    let result2 = sortFn (aList |> sortFn)
    result1 = result2

Check.Quick (``sort once should be same as sort twice``
  bubbleSort)
```

Listing 7: FsCheck: Applying sort function twice

3.4 Using F# in a company

F# is also one of the languages influenced by Haskell. Earlier mentioned Agda as well as Idris are used purely for research purposes. F#, on the other hand, is also used in various commercial business purposes, with its popularity constantly increasing. According to the 2016 StackOverflow Developer Survey, it came out as the most paid tech worldwide.[37] F# is 69th on the TIOBE Programming Community index for September 2018.[29]

3.4.1 SWOT Analysis

3.4.1.1 Strengths

F# has behind itself Microsoft, the prominent and well-known corporation that does provide some pressure so that the language would spread more. The .NET platform is well-known and has a large user base. F# developers

can use .NET core, which is very fast and carefully developed. F# applications can run on Windows, Linux or Mac.

3.4.1.2 Weaknesses

All the while, F# remains language number two at Microsoft. First one is C#, and therefore it receives better tooling and better support. Tooling is steered by the community. It is in no way unified, and the contributors create the tools according to what they believe to be correct. It has its advantages, as in giving more freedom and more possibilities. It also has its disadvantages, such as not having unified tools and well-defined standards.

3.4.1.3 Opportunities

Company Jet.com which runs exclusively on F# was bought by Walmart. This gives exposure to the language. The frontend is another place where we can see some improvement and many opportunities. Frontend languages and the development tends to towards the functional style. This will be explained more into detail in the case study.

3.4.1.4 Threats

C# has already begun to implement certain features from the functional paradigm. For beginners, this can be an advantage; they can use the available features and may not feel the need to go to F#. There are also certain features that .NET community has gotten used to with C#, such as nameof and string interpolation, which are not available in F#. Luckily, F# does not have any other functional programming language that would be its opponent on the .NET platform.

3.4.2 Improved readability

In the first chapter I have written about operators that improve readability. The three most important ones are: pipe forward operator, pipe backward operator and composition operator. In the following examples, I will demonstrate how the mentioned operators improve code readability.

Pipe forward operator

We have a list of tuples where we want to extract only those where the first element is 1 and then increase the second element by one. Pipe forward operator |> is used to pass the intermediate result to the next function. List.map enables us to take a function and apply it to each element in the list and put the results into a new list. List.iter would do the same except for creating a new list. List.filter takes a Boolean condition as a parameter and produces a new list with the elements that satisfy the given condition.

```
let res =
    [(1,2);(1,3);(3,2);(4,3);(5,2)]
    |> List.filter(fun (a, b) -> a = 1)
    |> List.map(fun (a, b) -> (a, b + 1))
printfn "%A" res
```

Listing 8: Pipe forward operator

Pipe backward operator

Pipe backward operator takes a function on the left and applies it to the values on the right. Pipe backward operator may seem unnecessary, but it plays an important role in improving the code readability. It enables us to change operator precedence without adding parentheses.

Instead of:

```
let addTwo (n : int) = n + 2
printf "%A" (addTwo 2)
```

Listing 9: Without the pipe backward operator

We can remove parentheses:

```
let addTwo (n : int) = n + 2
printf "%A" <| addTwo 2
```

Listing 10: With the pipe backward operator

Forward composition operator

Forward composition operator allows us to compose several functions into one another. There is also a backward composition operator, which is an inversion of the forward one.

```
let square a = a ** 2.0
let addFive n = n + 5.0
let squarePlusFive = square >> addFive
printf "%A" <| squarePlusFive 3.0
```

Listing 11: Forward composition operator

3.5 Haskell vs F#

When choosing the language for enterprise applications development, one needs to consider an array of criteria. In section Active development we analyze whether the language will still be there in ten years or is it just a fad, as well as how active the development of the language is and who is behind it. Are the language and the libraries well documented, is there a substantial base of libraries we can use in production? These will be answered in sections Language documentation and Library ecosystem. The answers to what kind of application domains is the language suitable for can be found in the Versatility section. After that, we will take a look at the features of the languages. In Development tools, we compare IDEs and other tools that the developers use daily. How will the developers adopt the language? Are there quality educational materials and what is the community like? These questions are also analyzed as they are also important factors when choosing a language for enterprise applications development.

3.5.1 Active development

When choosing a language for an enterprise application, it is important to consider whether language will still be there in a few years. An indication of this can be the number of people that are involved in language development. If there are a lot of people invested in the development, it may not necessarily guarantee better quality, but it is more likely that such a language will last longer. One example of a language that from a technical point of view looks very promising and exciting, but because only one person is working on its development is Purescript. This can dissuade companies from using such a language. Investing their time and money into working on a project that is

developed in a language that may not continue its development or may not even be there in a few years, does not sound tempting.

F# started in Microsoft Labs and was a Microsofts product. In 2013 they had decided to go open source and created the F# Software Foundation. It maintains the core open-source F# code repository. They maintain F# compiler, F# language specification, the F# core library, and the tools. F# Software Foundation acknowledges the role of Microsoft as the current primary developer of the language.[38] The F# Compiler, Core Library & Tools is the GitHub repository maintained by the F# Software Foundation. It has 179 contributors.[39] The F# Language, Library, and Visual F# Tools Repository is the GitHub repository maintained by Microsoft. All of the changes made here are eventually propagated to all packagings in F#. It has 144 contributors.[40] At the source tree of The Glasgow Haskell Compiler, we can see that they currently have 465 contributors. GHC has a large number of volunteer contributors. [41]

F# is the only functional language on the .NET platform, and that has its advantages. Even though Microsoft still favors C# over F#, and that can also be seen in terms of C# language getting upgraded tools and new libraries before F#. It is also more actively developed. For Microsoft, F# is still the language number two, but in terms of functional languages, it is number one. Haskell has an enthusiastic community, which can be seen from a much larger base of contributors. Neither of these languages can be seen as a fad, and it is very likely both of them will continue to grow and will stay here for a long time.

3.5.2 Language documentation

F# language reference provides a comprehensive and easy to read explanation of core language concepts and keywords.[6] On the other hand, official Haskell documentation provides links to many other websites. Therefore, the user needs to gather data from various sources to get the full picture. It can be cumbersome for first-time users to start using the language if it is difficult to find a place where the basic concepts of the language are well explained. The Haskell community does help a great deal since they have put together Haskell Wiki which has explanations of many core concepts. Unfortunately, the user needs to know what they are looking for to find it.[1]

I would say that when comparing these two, we see a significant difference. Microsoft provides the standard, straightforward documentation for F#. Haskell provides links to numerous tutorials, most of which are created by the passionate users of Haskell. Both seem sufficient, but F# seems more intuitive and more beginner friendly.

3.5.3 Library ecosystem

The following data is gathered from the website Libraries.io, which monitors open source packages across different package managers. There is a total of 1,806 packages in F#. There are over 8,488 packages available in Haskell. That is almost five times more packages than in F#. It is an indicator that the Haskell community is more active. It is also an indicator of how likely you will be to find the package that could help you in developing your project. Even when taking into account that Haskell has been around for 15 years before F#, hypothetically Haskell developers have been producing twice as many packages per year compared to F#. Of course, the course of development of these packages would need to be taken into account. Maybe Haskell package production was also slower in the beginning and received a boost in the meantime. Taking that into account, the same thing may happen with F# as well.[42]

It is important to bear in mind that a bigger number of packages does not necessarily mean better quality. Haskell does win size wise, but the quality of its packages compared to F# would be difficult to measure.

3.5.4 Versatility

Both languages are used in many different types of applications. They excel at some domains and tasks, are good just enough for others and for some domains they are not suitable at all. A versatile language can be beneficial to a programmer since they can use their knowledge for different programming domains.

When it comes to writing a compiler, maintenance, single-machine concurrency, type-driven development, and parsing, Haskell is considered to be the best language for it.[43] According to Microsoft, F# *has particular strengths in data-oriented programming, parallel I/O programming, parallel CPU programming, scripting, and algorithmic development.*[44] Haskell is not suitable for mobile development. On the other hand, F# is supported by Xamarin. Xamarin is a platform that lets you create native Android, iOS and Windows mobile applications with a single .NET code base. Thanks to this, developers can use F# to build mobile applications.[45]

Mobile applications are in great need nowadays, and the majority of enterprise applications are expected to have a mobile version as well. This is what gives F# an advantage over Haskell. But this can be considered only a small part of the whole enterprise applications development process and may not always play an important role. Both languages have their strengths and weaknesses, and according to what one is building, both choices should be considered.

3.5.5 Evaluation model

Previously, we have mentioned the advantages and the disadvantages of lazy evaluation, the former being not evaluating unless necessary and the latter less predictability. F# is a strict language, but we can use the keyword lazy to create a lazy expression. Haskell is a lazy language by default. Turning off laziness is not as straightforward as using a keyword, but there are ways to go around it. One example is enabling the Strict module in your code, which then switches functions to be strict by default, also allowing for optional laziness.[1]

3.5.6 Development tools

Visual Studio is a fully-featured integrated development environment (IDE) on Windows for building every type of .NET application, including F#. There is also Visual Studio for Mac for building native Android, iOS, macOS, and Windows apps with Xamarin. Visual Studio Code runs not only on Windows and macOS but also on Linux. There is a specific Ionide F# extension for Visual Studio Code, as well as Atom.

Linter (or just lint) is a tool that analyzes source code to highlight programming errors, bugs, and stylistic errors. HLint is such a tool for Haskell that provides suggestions when writing code. It can be integrated with different IDEs, such as Emacs or Atom. There are also linter tools for F#, although not widely used. There are numerous IDEs with packages specifically for Haskell such as KDevelop, Visual Studio Code, and Vim. Leksah is an IDE written in Haskell for Haskell.

Both programming languages have not only sufficiently developed tools, but also many options for the developer to choose.

3.5.7 Adopting the language

F# remains a language relatively unheard of, whereas there is an increasing number of universities that are adding Haskell to their curriculum. Even so, Haskell is considered to have a steeper learning curve. For experienced C# developers transition to F# is easy. Because of interoperability with C#, companies can still use the existing code base, while allowing the developers to implement some parts in F#. Therefore for already established companies and the before said reasons using F# over Haskell would be an advantage. For startups that are yet to write their first programs, both choices would be just as good.

Both F# and Haskell have large communities interested in improving and spreading the languages. According to the data from the review, Haskell ranks first in terms of popularity. Although this should not be considered discouraging since Haskell is an older language and has had more time to gain its popularity. Behind F# we have Microsoft, which can be helpful for proving

to management (those deciding on which technologies will be used) that F# is not just a new fad that will be dead in a few years.

Despite the fact that F# has less popularity, it still may be a better choice for enterprise applications development in regards to language adoption. Haskell does not have a gradual path for migration, which may be great for learning functional programming, but is not advisable for enterprises. Because of that, F# is more suitable for adoption.[46]

3.5.8 Educational materials

Often mentioned Haskells steep learning curve might not be due to the difficulty of the language, but because it lacks quality learning materials. Haskell learning materials are not systematic, so one has to look at several different sources to find what they need.

F# allows a smoother transition (due to it not being purely functional), but this may mean that the developer will not be using pure functional constructs but rather often relying on the imperative programming style. This can lead to not taking complete advantage of the functional paradigm.

3.5.9 Community

F# first appeared in 2005, whereas Haskell first appeared in 1990. It gives Haskell 15 years of advantage in building a larger community and increasing its popularity. It is on the 40th place in TIOBE Programming Community index, while F# is on the 69th, falling 29 places under.

There are 45,337 repository results with tag Haskell in GitHub, compared to meager 15,703 repository results when searching for tag F#.[41] There are 40,417 questions on StackOverflow using tag Haskell, compared to 13,621 questions using tag F#.[47]

Both communities are very active, but Haskell is, simply put, more popular.

Chapter 4

Case study

4.1 Enterprise applications development

Enterprise applications are complex systems developed specifically for the needs of an enterprise. Enterprise applications will almost always involve two things: persistent, large amounts of data and business logic.

Persistent data is defined as data that does not change. The data usually needs to be available all the time, for the years to come. The data is accessed by multiple users with different level of clearance at the same time. The users can edit the data at any time. Therefore the data transactions and updates in the database need to be fast. This also leads to the need for multiple user interfaces. Data needs to be kept secure. Business rules are determined by the enterprise, and the application needs to behave in accordance with them. We need to consider the integration as well. Often, the application needs to interact with other, already existing systems.

Enterprise applications need to be scalable. In case the enterprise encounters a rapid growth in either the number of users or the amount of data, the application should be able to support this. It needs to be able to overcome or to quickly retrieve from any kind of malicious (intentional or not) behavior it may come across. All of this leads to increased complexity.[48]

4.2 Real world application parts

In the following pages, we will look at certain parts of an application developed in F#. The parts we will look at are those that are used repeatedly in different enterprise applications. We will explain the usage, the implementation and the hypothetical Haskell implementation and comment on the differences.

4.2.1 Event sourcing

One of the components that almost every application will have is event sourcing. We have talked about immutability before and event sourcing is a great example of it being done properly and the advantages it can provide us with. Event Sourcing is storing the system changes itself instead of result of such changes. Instead of storing the current state of an object each time an update is made, we store the fact that there was an event making an update. In order to get the current state, we go through the sequence of events and apply all of them one by one. This way, we can for example go to a certain date and look at the state of the application at that time.

We can look at the following example to understand the advantages of event sourcing. Customer has an e-shop application and just like any other regular one, there is the basket to which you add your items. You can add them, as well as remove them from the basket and add them later. Marketing can come up with the idea that in order to improve sales, they would like to know how much time is spent between the customer adding and removing the item from the basket or vice versa. Instead of just knowing the current state, that is whether the item is in the basket or not, with event sourcing we can also take a look at the events from the history and create insights. This is often added to application, even without customer explicitly asking for it, because in the future they may become interested in things they were not before.

One of the disadvantages is that when there is a large number of changes, this means there will be a long sequence of events that the application needs to go through. Usually, this is solved by deciding at a certain moment to only go a certain number of events back, or always start from a certain event. This way, we can improve the performance. Another thing to bear in mind is that the complexity of application. Compared to simple CRUD((create, read, update, delete) against SQL table it is more complex to implement.

4.2.1.1 F#

In the following snippets I will demonstrate how Event Sourcing works in F#. We will take a look at a simple example application of a basket in an e-shop.

Command is an instruction to your system than can lead to new Events.

```
type Command =
    | AddProduct of CmdArgs.AddProduct
    | RemoveProduct of CmdArgs.RemoveProduct
    | RemoveAllProducts
    | OrderBasket of CmdArgs.OrderBasket
```

Listing 12: Event Sourcing: Command Union Type

Because each case contains event-specific data it needs to be defined in a different module:

```
module CmdArgs =
    type AddProduct = {
        Id : int
        Name : string
        Price : int
    }
    type RemoveProduct = {
        Id : int
    }
    type OrderBasket = {
        Id : int
        Price : int
        DeliveryDate : DateTime option
    }
```

Listing 13: Event-specific data defined

Event is fact that happened in past. One more thing we can notice here is the domain driven design approach. The verbs are now in the past tense because they are part of the ubiquitous language. Ubiquitous language is a term Eric Evans uses in Domain Driven Design, explaining that the business domain language should be used throughout the all development process and even in the source code, which is our example here.

```
type Event =
    | AddedProduct of CmdArgs.AddProduct
    | RemovedProduct of CmdArgs.RemoveProduct
    | RemovedAllProducts
    | BasketOrdered of CmdArgs.OrderBasket
```

Listing 14: Event Sourcing: Event Union Type

State is structural view of impact of Events.

```
type Product = {
    Id : int
    Name : string
    Price : int
    isSold : bool
}
type State = {
    Products : Product list
}
    with static member Init = { Products = [] }
```

Listing 15: Event Sourcing: State

Execute function is the place where Events are created based on Commands. This function can throw an exception.

The function signature is

$$'state -> ' command -> ' eventlist$$

```
let execute state command =
    let event =
        match command with
        | AddedProduct args ->
            args.Id
            |> onlyIfNotAdded state
            |> (fun _ -> AddedProduct args)
        | RemovedProduct args ->
            args.Id
            |> onlyIfAdded state
            |> (fun _ -> RemovedProduct args)
        | RemoveAllProducts -> RemovedAllProducts
        | OrderBasket args ->
            args.Id
            |> onlyIfNotEmpty state
            |> (fun _ -> BasketOrdered args)
    event |> List.singleton //we must return list of events
```

Listing 16: Event Sourcing: Execute function

Defining the validation functions.

```
let onlyIfNotAdded (state:State) i =
    match state.Products
          |> List.tryFind (fun x -> x.Id = i) with
    | Some _ -> failwith "Product already added"
    | None -> state

let onlyIfAdded (state:State) i =
    match state.Products
          |> List.tryFind (fun x -> x.Id = i) with
    | Some product -> product
    | None -> failwith "Product not added"

let onlyIfNotEmpty (state:State) =
    match state.Products
          |> List.isEmpty with
    | true -> failwith "No products in basket"
    | false -> state
```

Listing 17: Event Sourcing: Validation functions

In apply function we affect the State by applying each change on State. This function must never throw an exception. The signature of Apply function is

$$'state -> 'event -> 'state$$

```
let apply state event =
    match event with
    | AddedProduct args ->
        let added = {
            Id = args.Id
            Name = args.Name
            Price = args.Price
            isSold = false
        }
        { state with Products = added :: state.Products}
    | RemovedProduct args ->
        { state with Products = state.Products
                     |> List.filter
                        (fun x -> x.Id <> args.Id) }
```

Listing 18: Event Sourcing: Apply function

```
    | RemovedAllProducts -> { state with Products = [] }
    | BasketOrdered args ->
        let product =
            state.Products
            |> List.find (fun x -> x.Id = args.Id)
            |> (fun t -> { t with isSold = true })
        let otherProducts = state.Products
                            |> List.filter
                               (fun x -> x.Id <> args.Id)
        { state with Products = product :: otherProducts }
```

Listing 19: Event Sourcing: Apply function

Event store is a special, append-only database designed for storing (appending) Events. One can either implement their own or use an already existing library. Event Store or In this example, we will use CosmoStore which is an F# Event Store library for various storage providers.

```
let store = CosmoStore.CosmosDb.EventStore.getEventStore myC
```

Listing 20: Create Event Store

Event Store Command Handler has two dependencies partially applied as parameters: Event Store implementation and Aggregate type with Init, Execute and Apply functions. In the following snippet, we define the Aggregate type.

```
type Aggregate<'state, 'command, 'event> = {
    Init : 'state
    Apply: 'state -> 'event -> 'state
    Execute: 'state -> 'command -> 'event list
}

let productsAggregate = {
    Init = State.Init
    Execute = execute
    Apply = apply
}
```

Listing 21: Aggregate type with Init, Execute and Apply functions

```
let getCurrentState () =
    store.GetEvents "Products" EventsReadRange.AllEvents
    |> Async.AwaitProduct
    |> Async.RunSynchronously
    |> List.map (fun x ->
                 Mapping.toDomainEvent (x.Name, x.Data))
    |> List.fold productsAggregate.Apply
                 productsAggregate.Init

let append events =
    events
    |> List.map Mapping.toStoredEvent
    |> List.map (fun (name,data) ->
                 { Id = Guid.NewGuid();
                   CorrelationId = Guid.NewGuid();
                   Name = name; Data = data;
                   Metadata = None })
    |> store.AppendEvents "Products" ExpectedPosition.Any
    |> Async.AwaitProduct
    |> Async.RunSynchronously
    |> ignore

let handleCommand command =
    let currentState = getCurrentState()
    let newEvents = command
                    |> productsAggregate.Execute currentState
    newEvents |> append
    newEvents
```

Listing 22: Event Sourcing: Apply function

4.2.1.2 Haskell

In the following snippets I will demonstrate how Event Sourcing works in Haskell.

```haskell
data AddProduct
    = AddProduct
    { addProductId :: Int
    , addProductName :: String
    , addProductPrice :: Int
    } deriving (Generic, Show, ToJSON, FromJSON)

data RemoveProduct
    = RemoveProduct
    { removeProductId :: Int
    } deriving (Generic, Show, ToJSON, FromJSON)

data OrderBasket
    = OrderBasket
    { orderBasketId :: Int
    , orderBasketPrice :: Int
    , orderBasketDeliveryDate :: Maybe UTCTime
    } deriving (Generic, Show, ToJSON, FromJSON)
```

Listing 23: Entities

```haskell
data Command
    = AddProductCmd AddProduct
    | RemoveProductCmd RemoveProduct
    | RemoveAllProductsCmd
    | OrderBasketCmd OrderBasket
```

Listing 24: Command

```haskell
data Event
    = AddedProduct AddProduct
    | RemovedProduct RemoveProduct
    | RemovedAllProducts
    | BasketOrdered OrderBasket
    deriving (Generic, Show, ToJSON, FromJSON)
```

Listing 25: Event

```haskell
data Product
    = Product
    { productId :: Int
    , productName :: String
    , productPrice :: Int
    , productIsSold :: Bool } deriving Show

data State
    = State
    { stateProducts :: [Product] } deriving Show

state0 :: State
state0 = State []
```

Listing 26: State

```haskell
execute :: State -> Command -> IO [Event]
execute state cmd = case cmd of

    AddProductCmd args -> do
        let pid = addProductId args
        let products = stateProducts state
        case find (\p -> productId p == pid) products of
            Just _ -> error "Product already added"
            Nothing -> pure [AddedProduct args]

    RemoveProductCmd args -> do
        let pid = removeProductId args
        let products = stateProducts state
        case find (\p -> productId p == pid) products of
            Just _ -> pure [RemovedProduct args]
            Nothing -> error "Product not added"

    RemoveAllProductsCmd -> pure [RemovedAllProducts]

    OrderBasketCmd args -> do
        let products = stateProducts state
        case products of
            [] -> error "No products in basket"
            _ -> pure [BasketOrdered args]
```

Listing 27: Execute function

```
apply :: State -> Event -> State
apply state evt = case evt of
AddedProduct args ->
    let addedProduct
          = Product
          { productId = addProductId args
          , productName = addProductName args
          , productPrice = addProductPrice args
          , productIsSold = False
          }
  in state {stateProducts=addedProduct:(stateProducts state)}
RemovedProduct args -> let pid = removeProductId args
                       in state { stateProducts = filter
                            (\p -> productId p /= pid)
                            (stateProducts state) }
RemovedAllProducts -> state { stateProducts = [] }
BasketOrdered args ->
    let pid = orderBasketId args
        newProducts =  map (\p -> if productId p == pid
                            then p { productIsSold = True }
                            else p) (stateProducts state)
    in state { stateProducts = newProducts }
```

Listing 28: Apply function

```
data Aggregate state command event
    = Aggregate
    { aggregateInit :: state
    , aggregateApply :: state -> event -> state
    , aggregateExecute :: state -> command -> IO [event]
    }

productsAggregate :: Aggregate State Command Event
productsAggregate
    = Aggregate
    { aggregateInit = state0
    , aggregateApply = apply
    , aggregateExecute = execute
    }
```

Listing 29: Aggregate

```
productsStreamName :: Text
productsStreamName = "products"

eventName :: Event -> Text
eventName (AddedProduct _) = "added-product"
eventName (RemovedProduct _) = "removed-product"
eventName (RemovedAllProducts) = "removed-all-products"
eventName (BasketOrdered _) = "basket-ordered"

getCurrentState :: Connection -> IO State
getCurrentState conn = do
    let eventStoreStream = throwOnError
      (readThroughForward conn
      (StreamName productsStreamName)
      ResolveLink streamStart Nothing Nothing)
    let productEventsStream =
      S.map ((fromJust . recordedEventDataAsJson) .
        fromJust . resolvedEventRecord) eventStoreStream
    S.fold_ (aggregateApply productsAggregate)
      state0 id productEventsStream

append :: Connection -> [Event] -> IO WriteResult
append conn evts = do
        let eventStoreEvents = map toEventStoreEvt evts
        wait =<< sendEvents conn
          (StreamName productsStreamName)
        anyVersion eventStoreEvents Nothing
    where
        toEventStoreEvt evt =
            let evtName = eventName evt
            in createEvent (UserDefined evtName)
              Nothing (withJson evt)

handleCommand :: Connection -> Command -> IO [Event]
handleCommand conn cmd = do
    currentState <- getCurrentState conn
    newEvents <- (aggregateExecute productsAggregate)
      currentState cmd
    append conn newEvents
    pure newEvents
```

Listing 30: Event store operations

4.2.1.3 Comparison

Due to the fact that Haskell is a pure language, by looking at the implementation of event sourcing in both of the languages, this is where Haskell wins. Since F# is not a pure functional language, it is up to the programmer to ensure referential transparency.

The Haskell code has been written with the goal of being as close as possible to the F# source. As we can see the difference is minimal. A big difference can be spotted on the 'apply' function. While the F# implementation states that its type is 'state -> event -> state' and that it shall not throw an exception, this is not exactly accurate. The 'apply' function is actually not supposed to perform any side effects, i.e., the computation for the following state should be its only purpose, and it should use only its inputs to accomplish this task. The reason for this is easy to understand. An implementation that would fail to satisfy this property could compute a different sequence of states when applied to the same sequence of events. In many cases, the events would then not even be possible to apply as they are consequences of the previous states.

The same function signature in Haskell has a very different meaning. The type 'state -> event -> state' itself constrains the implementation to satisfy the above-listed properties. If it wanted to perform any side-effects, this would have to be expressed in the return type. The 'execute' function which is free to perform any side-effects returns a value of type 'IO [Event]' which signals that, while the result of running such a program is a list of 'Event's it is indeed an effect-full operation.

This would guarantee that even a new team-member would not by accident start performing effectfull operations as inserting any function querying the world state would result in a compilation error.

4.2.2 Frontend

An important part of an application, furthermore a crucial one in certain applications is frontend.

4.2.2.1 F#

Fable is a compiler from F# to JavaScript. It has so-called Elmish style, meaning it takes certain characteristics from Elm. It follows the model view update style of architecture, that was made famous by Elm.

How does it work? We have two data types, one of them is the union type of message, and the other is a model that is influenced by the message type. We always start with some defined model. Further on, we have two functions that work with them. The first one is the *update* function, it has a model, it takes the message as the parameter, and it returns a model. Its only concern

is to update the model based on the message type that it receives. The second function is the *view* function that receives the model and returns HTML.

This is a demonstration of the immutability principle, we do not change anything in the template, we get some model, and then we call two functions (*update* and *view*) and based on them we will get the new HTML. If the *update* function has changed the model, then the *view* function will create the new HTML. This way we dont need to change HTML physically. We will always get a new version of HTML, a difference comparison will be made between the old and the new HTML, and the result of that will be displayed.

We will take a look at a simple application that receives input, and after clicking the button, it prints out the entered text.

```
module Elmish.SimpleInput
open Fable.Core.JsInterop
open Fable.Helpers.React
open Fable.Helpers.React.Props
open Elmish
open Fulma
```

Listing 31: Extensions needed for the Fable demo application

In the following snippet, we see two types: Model and Msg. Function init initializes the Model values. Function *update* matches (pattern matching in action) the received Msg and according to it acts.

```
type Model = { UserName : string; DisplayName : string }
type Msg =
    | SaveName of string
    | ShowName of string
let init () = { UserName = ""; DisplayName = ""}, Cmd.none
let update (msg:Msg) (model:Model) =
    match msg with
    | SaveName newName ->
        { UserName = newName; DisplayName = "" }, Cmd.none
    | ShowName newName ->
        {UserName = ""; DisplayName = newName}, Cmd.none
```

Listing 32: Model, Msg and Update function

The function *view* in the following snippet creates the HTML based on the Model.

```
let view model dispatch =
    div [ Class "main-container" ]
        [ input [ Class "input"
                  Value model.UserName
                  Placeholder "Enter your name"
                  OnChange (fun ev -> ev.target?value
                                      |> string
                                      |> SaveName
                                      |> dispatch) ]
          span [ ]
            [ str "Hello, "
              str model.DisplayName
              str "! " ]
          div [ Class "block" ]
                  [ button
                  [ OnClick (fun _ -> model.UserName
                                      |> ShowName
                                      |> dispatch) ]
                                      [ str "Click me" ] ]
                ]
```

Listing 33: View model function in Fable

```
Program.mkProgram init update view
|> Program.withConsoleTrace
|> Program.withReactSynchronous "elmish-app"
|> Program.run
```

Listing 34: Running the program

4.2.2.2 Haskell

Purescript is a strongly-typed functional programming language that compiles to JavaScript. One of its most famous libraries is Halogen. Creating application in Purescript assumes knowledge of advanced topics, such as monads and functors. Halogen needs to know how to effectively run the monad, which is expressed by its hoist function:

```
hoist
  :: forall h f i o m m'
```

Listing 35: Purescript example

We can see the large number of arguments that the function needs to take. This is just one of the examples of the increased complexity when writing an application in Halogen. The reason behind this is purity of the language.

4.2.2.3 Comparison

Using Fable is simple and straightforward. The developers can write backend in F# and use their existing knowledge to create frontend as well. Online REPL available on the Fable.io website is a significant advantage as well. Developers can try Fable out without having to download or set anything up themselves. Due to the fact that Haskell is a pure language, trying to create frontend in such a language leads to added complexities. This is why, when looking at the implementation of frontend in both of the languages we can see that F# is more suitable for it.

Conclusion

The goal of this thesis was to compare two functional languages Haskell and F# with regards to the enterprise applications development. I compared the two languages from a technical, as well as a business point of view. In the Analysis chapter, I looked at certain criteria that are taken into account when choosing a language for an enterprise project. In the Case study chapter I analyzed a real project written in F# and compared it with a hypothetical Haskell implementation commenting on the differences. Based on this work I concluded the following.

Haskell comes from the academic environment, and that is where it is still the most suitable. Haskell enforces purity that can be crucial in such applications as blockchain, but the majority of enterprise applications do not require such rigorosity. Not only do they not require it, but overcoming the steep learning curve and the transition process could be detrimental to the business. F# allows for a smoother transition, especially when taking into account its interoperability with C#. F# is developed and supported directly by Microsoft, and a great number of libraries and frameworks are intended for enterprise applications development.

It is important to carefully examine the characteristics of the enterprise project and based on them decide which language would be better for that specific project. If the benefits that Haskell will bring are greater than its disadvantages, as is the example with the blockchain technologies, then one should opt for Haskell. All of that being said and based on the review and the analysis, it seems that in most of the cases, F# is more suitable for enterprise applications development than Haskell.

Bibliography

[1] HaskellWiki [online]. [visited on 2019-28-03]. Available from: https://wiki.haskell.org/Haskell

[2] OKASAKI, C. Purely Functional Data Structures [online]. [visited on 2019-28-03]. Available from: http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

[3] SONDERGAARD, H.; SESTOFT, P. Referential transparency, Definiteness and Unfoldability [online]. [visited on 2019-28-03]. Available from: http://www.itu.dk/people/sestoft/papers/SondergaardSestoft1990.pdf

[4] HUDAK, P. Conception, evolution, and application of functional programming languages [online]. [visited on 2019-28-03]. Available from: https://dl.acm.org/citation.cfm?doid=72551.72554

[5] HUDAK, P.; HUGHES, J.; et al. A History of Haskell: Being Lazy With Class [online]. [visited on 2019-28-03]. Available from: http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf

[6] F# Language Reference. In: F# Guide [online]. [visited on 2019-28-03]. Available from: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/index

[7] Miranda [online]. [visited on 2019-28-03]. Available from: http://miranda.org.uk/

[8] MILNER, R. *The Definition of Standard ML (Revised)*. The MIT Press, second edition, ISBN -262-63181-4.

[9] MILNER, R.; TOFTE, M. Commentary on standard ML [online]. [visited on 2019-28-03]. Available from: https://www.semanticscholar.org

[10] HARPER, R. Programming in Standard ML. Available from: http://www.cs.cmu.edu/~rwh/isml/book.pdf

[11] BOVE, A.; DYBJER, P.; et al. A Brief Overview of Agda A Functional Language with Dependent Types [online]. [visited on 2019-28-03]. Available from: http://www.cse.chalmers.se/~ulfn/papers/tphols09/tutorial.pdf

[12] NORELL, U.; CHAPMAN, J. Dependently Typed Programming in Agda [online]. [visited on 2019-28-03]. Available from: http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf

[13] Documentation for the Idris Language [online]. [visited on 2019-28-03]. Available from: http://docs.idris-lang.org/en/latest/

[14] Unofficial FAQ Idris [online]. [visited on 2019-28-03]. Available from: https://github.com/idris-lang/Idris-dev/wiki/Unofficial-FAQ#why-isnt-idris-lazy

[15] MARLOW, S. Fighting spam with Haskell [online]. [visited on 2019-28-03]. Available from: https://code.fb.com/security/fighting-spam-with-haskell/

[16] POP, I. Experience Report: Haskell as a Reagent [online]. [visited on 2019-28-03]. Available from: https://k1024.org/papers/icfp10-haskell-reagent.pdf

[17] Introduction to F# & Essential Tools. In: The F# Survival Guide [online]. [visited on 2019-28-03]. Available from: https://web.archive.org/web/20110708212732/http://www.ctocorner.com/fsharp/book/ch2.aspx

[18] WLACSHIN, S. Seamless interoperation with .NET libraries. In: F# for fun and profit [online]. [visited on 2019-28-03]. Available from: https://fsharpforfunandprofit.com/

[19] Glasgow Haskell Compiler User's Guide [online]. [visited on 2019-28-03]. Available from: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/intro.html

[20] JONES, S. P. Beautiful concurrency [online]. [visited on 2019-28-03]. Available from: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf

[21] O'SULLIVAN, B.; STEWART, D.; et al. Software transactional memory. In: Real World Haskell [online]. [visited on 2019-28-03]. Available from: http://book.realworldhaskell.org/read/software-transactional-memory.html

[22] KHAN, M. E.; KHAN, F. Importance of Software Testing in Software Development Life Cycle [online]. [visited on 2019-28-03]. Available from: https://www.ijcsi.org/papers/IJCSI-11-2-2-120-123.pdf

[23] O'SULLIVAN, B.; STEWART, D.; et al. Testing and quality assurance. In: Real World Haskell [online]. [visited on 2019-28-03]. Available from: http://book.realworldhaskell.org/read/testing-and-quality-assurance.html

[24] QuickCheck: Automatic testing of Haskell programs [online]. [visited on 2019-28-03]. Available from: http://hackage.haskell.org/package/QuickCheck

[25] Bubble Sort in Haskell [online]. [visited on 2019-28-03]. Available from: https://gist.github.com/zooxyt/fbeb790f6c95fc0f78c1536a9af2beb2

[26] The GHCi Debugger [online]. [visited on 2019-28-03]. Available from: https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-debugger.html

[27] Debug.Trace [online]. [visited on 2019-28-03]. Available from: http://hackage.haskell.org/package/base-4.12.0.0/docs/Debug-Trace.html

[28] haddock: A documentation-generation tool for Haskell libraries [online]. [visited on 2019-28-03]. Available from: https://hackage.haskell.org/package/haddock

[29] TIOBE Index [online]. [visited on 2019-28-03]. Available from: https://www.tiobe.com/tiobe-index/

[30] FP Complete - Commercial Haskell Accelerator [online]. [visited on 2019-28-03]. Available from: https://haskell.fpcomplete.com/

[31] The commercial Haskell group [online]. [visited on 2019-28-03]. Available from: https://github.com/commercialhaskell/commercialhaskell#readme

[32] How much time have you invested in order to have a good grasp on Haskell? In: StackOverflow [online]. [visited on 2019-28-03]. Available from: https://stackoverflow.com/questions/3750529/how-much-time-have-you-invested-in-order-to-have-a-good-grasp-on-haskell

[33] StackOverflow Developer Survey Results 2018 [online]. [visited on 2018-28-12]. Available from: https://insights.stackoverflow.com/survey/2018/#overview

[34] FOWLER, M. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, second edition, ISBN 978-0134757599.

[35] COWIE, J. Detecting Bad Smells in Haskell [online]. [visited on 2019-28-03]. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.9950&rep=rep1&type=pdf

[36] HaRe wiki [online]. [visited on 2019-28-03]. Available from: https://github.com/RefactoringTools/HaRe/wiki

[37] StackOverflow Developer Survey Results 2016 [online]. [visited on 2018-28-12]. Available from: https://insights.stackoverflow.com/survey/2016

[38] F# Software Foundation [online]. [visited on 2019-28-03]. Available from: http://foundation.fsharp.org/

[39] The F# Compiler, Core Library & Tools (F# Software Foundation Repository) [online]. [visited on 2019-28-03]. Available from: https://github.com/fsharp/fsharp

[40] The F# compiler, FSharp.Core library, and tools for F# [online]. [visited on 2019-28-03]. Available from: https://github.com/Microsoft/visualfsharp

[41] GitHub [online]. [visited on 2019-28-03]. Available from: https://github.com/

[42] Libraries.io [online]. [visited on 2019-28-03]. Available from: https://libraries.io/

[43] State of the Haskell ecosystem. In: GitHub [online]. [visited on 2019-28-03]. Available from: https://github.com/Gabriel439/post-rfc/blob/master/sotu.md

[44] F# at Microsoft Research [online]. [visited on 2019-28-03]. Available from: https://www.microsoft.com/en-us/research/project/f-at-microsoft-research/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fcambridge%2Fprojects%2Ffsharp%2F

[45] Visual Studio Tools for Xamarin [online]. [visited on 2019-28-03]. Available from: https://visualstudio.microsoft.com/xamarin/

[46] WLACSHIN, S. Why F# is the best enterprise language. In: F# for fun and profit [online]. [visited on 2019-28-03]. Available from: https://fsharpforfunandprofit.com/

[47] StackOverflow [online]. [visited on 2019-28-03]. Available from: https://stackoverflow.com/tags

[48] FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, ISBN 9780133065213.

Appendix A

Acronyms

API Application Programming Interface

GHC Glasgow Haskell Compiler

HTML HyperText Markup Language

IDE Integrated development environment

ML Meta Language

OOP Object-oriented programming

REPL Read-eval-print loop

REST Representational State Transfer

STM Software Transactional Memory

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format