HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

# If Functional Programming Is So Great, Why Isn't Everyone Using It?

## Project Thesis 2

University of Applied Sciences Rapperswil

Spring Semester 2017


| | |
|---|---|
| Author: | Mario Meili |
| Advisor: | Prof. Dr. Farhad D. Mehta |
| Industry Partner: | Institute for Software (IFS) HSR |

## Abstract

*Functional programming has been considered to be an answer to many of the problems faced by software engineering today, especially in academic circles. Nevertheless, the large-scale adoption of functional programming in the mainstream of industry has been sluggish. Although there are a lot of different opinions about the underlying problem to this issue, it is hard to find literature that elaborates on the topic.*

*This report presents the results of a meta-study conducted on multiple aspects of the functional programming paradigm and its representing languages. It does so by identifying key issues responsible for the weak adoption of functional programming, and, more importantly, by proposing and evaluating measures that should positively influence the usage of functional programming languages in industrial, commercial and practical settings.*

**Acknowledgements**

I would like to thank my thesis advisor Prof. Dr. Farhad D. Mehta, who was always willing to lend a helping hand when I ran into problems. Whenever I felt lost, he pushed me in the right direction.

Also, I would like to express my gratitude to my girlfriend for providing me with unfailing support and continuous encouragement.

Finally, I would like to thank my colleague and friend Felix Morgner, who provided me with a GitHub API script for gathering repository data.

# Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All consulted sources are clearly mentioned and cited correctly. No copyright-protected materials are unauthorisedly used in this thesis.

Mario Meili

*Signature*

$27^{th}$ of July, 2017

*Date*

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**SCADA** Supervisory Control And Data Acquisition

**APIs** Application Programming Interfaces

**IDE** Integrated Development Environment

**IRC** Internet Relay Chat

**UI** User Interface

**NFRs** Non-Functional Requirements

**HTTP** Hypertext Transfer Protocol

**PLCs** Programmable Logic Controllers

**ERP** Enterprise Resource Planning

# 1. Introduction

Alonzo Church laid the groundwork for functional programming by introducing the lambda calculus in the 1930s [31]. Because of its Turing completeness, the lambda calculus can be thought of as a programming language itself, using function abstraction and application for computation [29]. Programming languages adopting the functional paradigm were first introduced in the late 1950s, when LISP was first described by John McCarthy [40]. Thereafter followed well known languages such as Algol 60 (1963), ISWIM (1966), Miranda (1986) and Haskell (1992) [51].

## 1.1. Problem

This means that functional programming languages have been around for over 50 years. But still, after all this time, the adoption of functional programming languages, especially in industrial projects, seems to be rather weak. But why was functional programming not adopted as widely as expected by the functional programming community? There are a lot of different opinions about regarding question, but no broad study looking at different aspects of the problem.

## 1.2. Goals

The aim of this thesis is to get rid of speculations and ill-founded opinions by

- systematically evaluating the extent to which functional programming is being used in various industrial, commercial and practical settings as well as the advantages and disadvantages of using functional programming in these settings,
- proposing and investigating reasons why the use of functional programming has been limited, especially in areas where its advantages are found to be clear
- and proposing, discussing and prioritising measures that could be taken to increase the benefit gained using functional programming in various industrial, commercial and practical settings.

## 1.3. Document Structure

The rest of the report is organised as follows. Chapter 2 introduces essential background concepts, as well as giving a brief overview of related work. Language adoption in general is discussed in Chapter 3. In Chapter 4, advantages and disadvantages of the functional programming paradigm are discussed based on theoretical and practical observations. Chapter 5 shows, how the universe of application areas can be divided into categories and their requirements. This allows to pinpoint possible application areas in a guided manner. The results are gathered and discussed in Chapter 6 and finally, Chapter 7 provides a conclusion on possible measures to take to increase the adoption of functional programming languages.

# 2. Background & Related Work

As already mentioned in Chapter 1, there are a lot of opinions about the advantages and disadvantages of the functional programming paradigm and functional programming languages. However, it is often the case that these opinions lack explanation and well founded justification. Section 2.1 lists exceptions to this circumstance. In Section 2.2, the core concepts of the functional programming paradigm are briefly discussed. This section may be skipped by programmers who are already experience in one or more functional programming languages.

## 2.1. Related Work

Backus wrote a paper for his Turing award lecture in 1978, in which he answers the question whether programming can be liberated from the von Neumann style [28]. In it, he describes all programming languages, which are not functional, as useless. Although amusing to read, the opinion of Backus is very biased and therefore unnecessarily harsh. Unfortunately, Backus argues solely on a technical level, completely ignoring social aspects of language adoption.
Later, in 1989, Hudak published a paper on the conception, evolution, and application of functional programming languages [36]. It is interesting to see, that 28 years ago, the myths around functional programming closely resemble the opinions of people today. In the introduction to the paper, Hudak writes:

> *Are functional languages toys? Or are they tools? Are they artifacts of theoretical fantasy or of visionary pragmatism? Will they ameliorate software woes or merely compound them?*

Some of these questions will be looked at closer in the following chapters of this thesis. But again, the content of the paper concentrates mainly on the features of the functional programming paradigm. Social aspects of language adoption are not covered. Also, the question of applicability for the industry is not answered.
In 1989, Hughes wrote one of the most famous papers about the importance of functional programming [37]. He argues that modularity is the key to successful programming and in order to be able to decompose problems into parts requires the ability to glue these parts together. In his opinion, higher-order functions and lazy

evaluation represent this perfect glue. Although the examples given in the paper seem reasonable, they are not very large in size and mostly of a mathematical nature. Therefore, the paper does not give an answer to the applicability of functional programming in industrial projects.
The book Real World Haskell [46] by O'Sullivan, Stewart and Goerzen presents applications solving real world problems written in the functional programming language Haskell. It is one of the few examples where an introduction to the language itself is left out and the application is put into the foreground. Unfortunately, the examples shown in the book do not make use of the newest version of the language and were never updated since 2008, when the book was released.
Meyerovich and Rabkin published a paper on principles for programming language adoption in 2012 [41]. They claim, that social sciences have studied adoption in many contexts and show how their findings are applicable to programming language design. They do not only state 15 hypotheses, but also raise 21 open-ended questions to be investigated in further research. Because the paper's main focus lies on the design of languages, it does not answer the stated problems of this thesis, but tries to provide the tools for designers of new languages, making these languages more likely for successful adoption.
Only one year later, in 2013, an empirical analysis of programming language adoption was also given by Meyerovich and Rabkin [42]. This paper summarises the results of a large study conducted using multiple large scale surveys and source repository meta data. It covers all aspects of adoption, including tooling, language features and even social aspects. It would be interesting to see how the statistics presented have changed in the last four years and also, an additional study with emphasis on functional programming languages might be rewarding. The results of this paper are discussed in more detail in Chapter 3.
At last, the papers [33] and [34] by Henderson, and the paper [50] by Treleaven discuss theoretical approaches to functional computer architectures and functional operating systems.

## 2.2. The Functional Programming Paradigm

In the following, the most important aspects of the functional programming paradigm are explained. Firstly, it is shown what it means for functions to be first class citizens of their corresponding language. Then, a definition for the term higher-order functions is given. After that, the concept of referential transparency is introduced, followed by the most important evaluation strategies. Finally, recursion and pattern matching are discussed. Since this section is intended for readers with little experience in functional programming, small example are provided, if suitable.

### 2.2.1. Functions as First-Class Citizens

Functional programming languages treat their functions as first class citizens. Functions treated this way are also called first-class functions. A function is said to be first-class, if it can be

- passed as an argument to other functions,
- returned from other functions,
- assigned to variables
- and, therefore, be stored in data structures.

Listing 2.1 shows an example in the programming language Haskell. The function `twice` is defined to take a function `f` and a value `a` as its arguments. It then applies the function `f` twice to the argument `x`, as can be seen on line 2.

```
1 twice :: (a -> a) -> a -> a
2 twice f x = f (f x)
3
4 fourTimes:: (a -> a) -> a -> a
5 fourTimes f x = twice (twice f) x
```

Listing 2.1: First-class functions in Haskell

On line 5, something interesting can be observed. The value of the subexpression `(twice f)` is actually just another function, that takes a value `x`. `twice` was partially applied in this scenario and this new function can be again passed as an argument for `twice`. The result is a new function `fourTimes`, which applies a function `f` four times to a value `x`. Note that this also means that anonymous functions can be defined and then stored in variables.

### 2.2.2. Higher-Order Functions

Higher-order functions are an essential part of functional programming languages. Actually, both functions defined in Listing 2.1 are higher-order functions, because they take a function as their argument. In fact, a function is called higher-order if one of the following holds:

- The function takes one or more functions as arguments.
- The function returns a function as its result.

In Listing 2.2, the definition of the function `flip` from the Haskell standard prelude is given. This function not only takes a function with two arguments as its argument, but also returns a new function, which in this case just flip the two arguments of the function argument.

```
1 flip :: (a -> b -> c) -> b -> a -> c
2 flip f y x = f x y
```

Listing 2.2: Higher-order function in Haskell

It is important to note that not all functions defined in a functional language are higher-order, but are always first-class citizens of the language.

### 2.2.3. Referential Transparency

When talking about the functional programming paradigm, it is usually assumed that there is no shared state in a program. This means that functions can never have side effects. If a functional programming language satisfies this criterion, it is called a pure functional programming language and referential transparency holds. It is defined as follows: Expressions are referentially transparent if they can be replaced with their corresponding values without changing the behaviour of the program they are part of. From that follows, that evaluating referential transparent functions always results in the same values for the same function arguments.

### 2.2.4. Evaluation Stratgies

Because of referential transparency, one has to consider the order in which expressions are evaluated. In fact, different functional programming languages come with different evaluation strategies. Thereby, the most important difference lies in evaluating in a strict order and evaluating in a lazy manner. Strict functional programming languages evaluate expressions as they occur, even if the result of the expression would only be needed to a later point in time, or, not at all. One such language is Standard ML.

```
1 fun f x = 17;
2 fun inf x = inf x;
```

Listing 2.3: Strict evaluation in SML

Listing 2.3 shows two function definitions in Standard ML. When trying to evaluate the expression `f (inf 0);` strictly, the result will be an endless recursion, because the function `inf` calls itself recursively, although function `f` does not care about its argument and could return `17`.

```
1 f :: a -> Int
2 f x = 17
3
4 inf :: a -> a -> a
5 inf x = inf x
```

Listing 2.4: Lazy evaluation in Haskell

In contrast to Standard ML, Haskell is a language that evaluates expressions lazily. Again, this means that expressions are only evaluated when and if their resulting value is needed. Listing 2.4 shows the same definitions of functions `f` and `inf`, but in the Haskell programming language. Now, when evaluating the expression `f (inf 0)`, the result is simply `17`. Haskell recognized, that an evaluation of the argument for function `f` is not necessary, therefore `inf` was never evaluated.

### 2.2.5. Recursion

Recursion is a technique not exclusive to functional programming. However, for functional programming languages, the significance of recursion is somewhat higher than for imperative languages. This is mainly because functional programming languages do not support constructs such as loops. Recursion can be used to achieve behaviour similar to loop iteration. In Listing 2.5, a recursive function is defined in Haskell.

### 2.2.6. Pattern Matching

Functional programming languages often make use of pattern matching. This means that a function can be defined multiple times, matching different patterns in their arguments.

```
1 sumlist :: Num a => [a] -> a
2 sumlist []       = 0
3 sumlist (x : xs) = x + sumlist xs
```

Listing 2.5: Pattern matching & recursion in Haskell

Listing 2.5 shows how the function `sumlist` can be defined by matching on the base case of the recursion, namely if applied to the empty list, and in addition on the recursive call, if the list is non-empty. As can be seen, pattern matching makes defining recursive functions straightforward and therefore quite simple.

# 3. Programming Language Adoption

In order to discuss the weak adoption of functional programming languages in the industry, it is important to first have a look at language adoption in general. This chapter aims to identify the main reasons of successful language adoption. The results presented mostly stem from [42], a paper summarising results of a large scale empirical analysis of programming language adoption conducted by Meyerovich and Rabkin. For gathering data, they combined survey research and software repository mining. Further referencing of this paper is omitted in the rest of this chapter.

## 3.1. Distribution of Language Popularity

When looking at the distribution of usage across different languages, Meyerovich and Rabkin found out, that a small number of programming languages account for the majority of language use. The following results were gathered by mining SourceForge[1] repository metadata between the years 2000 and 2010:

- The top 6 languages (most used in projects) account for 75% of all projects.
- The top 20 languages (most used in projects) account for 95% of all projects.
- General purposes languages compromise most of the top 20 languages (most used in projects).

These numbers make clear how hard it is for a programming language to become widely used. Section 3.2 shows that pushing aside an already established language is a difficult task, which is why this result is even worse for new languages.

## 3.2. Choosing a Language

One might think that the choice of language for new projects solely depends on the features provided by said language. Languages offering features suitable for the problem domain should be preferred to other languages. However, the evaluation of

[1] `https://sourceforge.net`

several large surveys revealed that intrinsic language features have only secondary importance. Instead, the factors influencing a developers choice of language can be summarized as follows:

- Open source libraries, existing code, and experience strongly influence a developers choice.
- Developers tend to stick to clusters of languages. And often, they stick to the same language they used for previous projects.
- Which languages developers learn is influenced by their education.
- Employees at larger companies place significantly more value on existing code bases and knowledge than do employees at small companies.

All these circumstances lead to widely used programming languages keeping their influence. Languages with low adoption are not likely to suddenly gain more interest. When looking at language features, the results look as follows:

- Developers generally value expressiveness and speed of development over language-enforced correctness.
- Developers see more value in unit tests than strong type systems for debugging.

Concluding this section, it can be said that the best chance to widen the adoption of a language is to develop high-value open source libraries.

## 3.3. Language Statistics

The previous sections of this chapter summarized general criteria that influence language adoption. The intention of this section is to provide a concrete prospect of how weak functional programming languages are adopted to the reader.
So far, the measure for determining the top languages has been the number of repositories a language has been used in. The TIOBE index for language popularity follows another approach. 25 selected search engines count the hits for searches on a large list of programming languages. The precise index definition can be found at [17]. Table 3.1 shows an excerpt of TIOBE index ranking for July 2017 [16].
It can be seen that languages only offering the functional paradigm first appear on rank 32, namely the language Lisp. However, there are a lot of multi-paradigm languages including functional features on top ranks in the list. It would be interesting to research the development of the ranking over time, but TIOBE does not give away their data freely.

| Rank | Programming Language | Paradigm |
|---|---|---|
| 1 | Java | Object-oriented (OO) |
| 2 | C | Imperative |
| 3 | C++ | Multi-paradigm (including functional, OO) |
| 4 | Python | Multi-paradigm (including functional, OO) |
| 5 | C# | Multi-paradigm (including functional, OO) |
| 8 | JavaScript | Multi-paradigm (including functional, OO) |
| 12 | Swift | Multi-paradigm (including functional, OO) |
| 15 | R | Multi-paradigm (including functional, OO) |
| 17 | MATLAB | Multi-paradigm (including functional, OO) |
| 30 | Scala | Multi-paradigm (including functional, OO) |
| 32 | Lisp | Functional |
| 40 | F# | Functional |
| 41 | Haskell | Functional |
| 43 | Erlang | Functional |
| 44 | Scheme | Functional |

Table 3.1.: TIOBE index ranking 2017

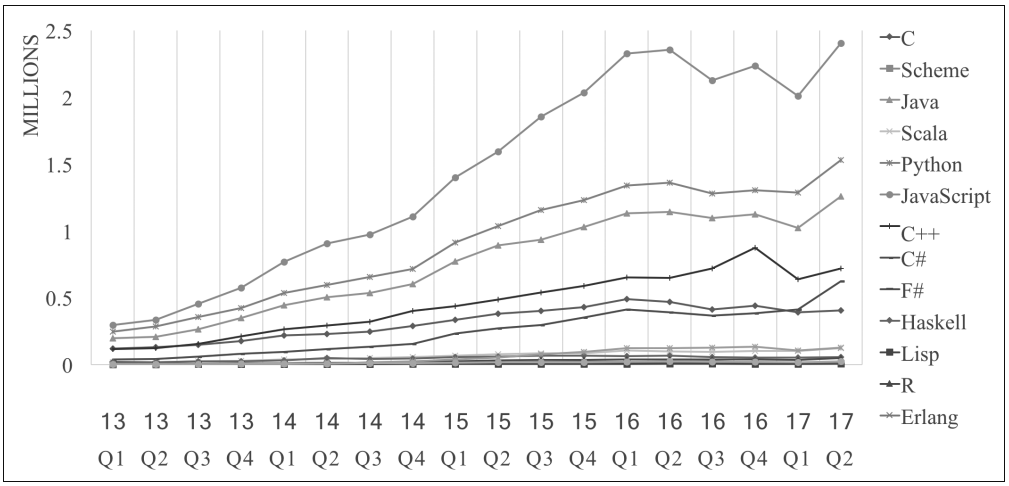


Figure 3.1.: Active GitHub repositories per quarter

GitHub[2] has been chosen as the data source for the statistic provided in Figure 3.1. The complete data set can be found in appendix A. The graph shows the active repositories of the above listed programming languages over time. A repository is regarded as active, if in the given time frame either a push, a pull request or issue activity took place. The clear leader is JavaScript with almost 2.5 million active repositories. What follows is a huge gap until, finally, the functional programming languages form the thick line on the bottom of the graph.

[2] `https://github.com`

# 4. Aspects Considered

Although it was shown in Chapter 3 that intrinsic language features have only limited effects on language adoption, the discussion thereof is necessary to narrow the field of possible application areas. This is, after all, the main aim of this thesis. In order to identify the most important advantages and disadvantages of the functional programming paradigm, several blog posts, e.g. [6, 8, 9, 13, 19] and [22], and forum board discussions, e.g. [14, 18] and [24] have been analyzed for the most prominent opinions. The results listed below were then taken as hypotheses for further evaluation:

- Functional programming languages perform poorly in real world (and also real-time) applications.
- Functional programming languages are predestined for parallelization.
- Functional programming is lacking the critical mass to be relevant.
- The tooling/library/infrastructure support for functional programming is lacking.

During the period of this thesis, a short survey has been conducted, asking developers from different backgrounds about their opinions on the statements given above. In total, 43 responses have been recorded from participants between the ages of 20 and 62. 62.8% of the participants use functional programming languages on a daily basis and another 23.3% make use of them weekly. The raw data of the survey can be found in appendix B. Survey results are presented in grey boxes.

Each section in this chapter discusses one of the hypotheses by firstly presenting the corresponding results of the survey and secondly analyzing the responsible paradigm aspect theoretically and practically.

## 4.1. Performance

Performance can be measured in two different ways. One is the number of instructions necessary to perform a task. The other one is memory consumption. Both can

be analyzed looking at data structures. Parallelization and concurrency obviously have a direct impact on performance, but this aspect is discussed separately in Section 4.2. Also, the order of evaluation plays and important role, as was discussed in Subsection 2.2.4. Because of that, the functional languages Standard ML and Haskell have been chosen as representatives. Standard ML is evaluating expressions in a strict manner, whereas Haskell uses the lazy evaluation strategy.

### 4.1.1. Operations on Persistent Data Structures

In purely functional programming languages, data structures are called persistent, which means they are immutable. The consequence is, that every update to a data structure results in copying the old data structure with the change and assign the newly created data structure in a variable.

For the survey conducted during this thesis, participants were asked to answer, if immutable data structure operations tend to run faster, slower or equally as fast as those of corresponding imperative implementations. The result is somewhat surprising. 39.5% of the participants thought that operations on immutable data structures were slower than on mutable ones. 37.2% believe that it makes no difference and 23.3% think that immutable data structure operations are faster. This is surprising because there is no leaning towards one answer. Should this not be clear?

#### In Theory

There are mutable data structures which allow update operations in constant time. One such example is dictionaries or, as they are also called, hash tables. Operations such as lookup, insertion and deletion of an element can be achieved in $O(1)$ (amortized runtime), because of in-place updates [47].
Obviously, this can not be achieved with immutable data structures. Also, dictionaries and arrays are two of the most widely used data structures there are. Purely functional data structures perform at most as good as their non-functional counterpart. As soon as in-place updates and memory references come into play, functional data structures place second.
Interestingly, quite some attempts were made to implement purely functional data structures in a way to get near non-functional ones regarding operation costs. Okasaki published [45], a book containing improved implementations of well-known data structures in the language Standard ML. Adams wrote [26, 27], where he shows how sets can be implemented more efficiently in functional languages, using a binary search tree of bounded balance as the basis [44]. In fact, trees are the most efficient approach to better operation costs of immutable data structures.

This means that the lower bound for operations lies at $O(log(n))$, where $n$ is the number of elements in the data structure.

#### In Practice

Both Standard ML and Haskell would probably not be accepted as usable programming languages, if they did not solve this obvious problem. Haskell makes use of monads, which allow the definition and use of mutable data structures without destroying functional properties outside of the monadic construct the data structure is built in [52]. Monads ensure that certain operations, which can have side effects, are executed in sequence. Note that this is no longer part of the purely functional programming paradigm, where side effects are not allowed.
Standard ML uses a construct called reference types, which may be updated and therefore allow the definition of mutable data structures similar to how monads do in Haskell [39]. This technique works only in languages following a strict evaluation strategy. This makes it unusable for the lazy language Haskell. As with mondas, this is a violation of the pure functional paradigm.
Side effects are necessary to write usable programs. Without them, programs could not perform input and output operations. Since monads and reference types can not only be used to define mutable data structures, but also to encapsulate all program behaviour with side effects, they can be viewed as a necessary evil. And in practice, they make sure that functional programming languages stay competitive in terms of performance.

### 4.1.2. Memory Consumption

Because data structures in functional programming languages are persistent, they need to be saved from anew, every time an update takes place. The old data structure stays assigned to its variable.

49.8% of the participants of the survey think, that this is the reason for immutable data structures consuming more memory than mutable ones. 41.9% voted for an equal amount of memory use and 9.3% have the opinion that less memory is used by immutable data structures. Interestingly, when looking at the memory consumption of functional versus imperative programs in general, the numbers are 67.4% for more, 27.9% for equally as much and 4.7% for less memory consumption.

#### In Theory

Memory consumption in purely functional languages is higher than in imperative ones. Every newly created value has to be assigned, if it should be available in multiple places. Reassignment of variables is not allowed, so every time a variable is assigned, more memory has to be allocated. Immutable data structures are especially bad, because the entire data structure has to be copied for every ever so small update operation. Theoretically, there is no doubt that functional programs must allocate more memory than imperative counterparts.

#### In Practice

As was already explained in Subsection 4.1.1, functional programming languages use constructs such as monads and reference types to be able to make use of mutable data structures. But for this to be effective, programmers have to be aware of what they are doing. Especially beginners will most likely not think about memory management at first. As for Haskell, monads introduce an entirely different syntax and are mostly learned in a late stage.
Haskell partially solves this problem using the technique of garbage collection. The Haskell wiki page on memory management of the Glasgow Haskell compiler [10] states the following:

> *Haskell computations produce a lot of memory garbage - much more than conventional imperative languages. It's because data are immutable so the only way to store every next operation's result is to create new values. In particular, every iteration of a recursive computation creates a new value. But GHC is able to efficiently manage garbage collection, so it's not uncommon to produce 1gb of data per second (most part of which will be garbage collected immediately).*

Standard ML also uses garbage collection to get rid of no longer used values. Summarising, it can be stated that the excessive use of memory is being dealt with. Although the memory is freed up by garbage collection, functional programming languages experience a setback performance wise.

### 4.1.3. Sidenote on Lazy Evaluation

So far in this section, a dark secret was kept behind the scenes. Namely, the impact of lazy evaluation on predicting performance was left out. The reason is simple. The impact of lazy evaluation on performance, both speed and memory wise, cannot be predicted. Evaluating expressions lazyily can, for example, lead to an expression being evaluated many times, if evaluation is deferred to a later point in time. This

can be prevented with a technique called sharing. Keeping only a single copy of an argument expression, and maintaining a pointer to it for each corresponding formal parameter, allows to evaluate said expression once, and replacing it by its value. It can then later be accessed through the pointers. It is not hard to see that this approach has a negative influence on memory consumption.

A question in the survey was for participants to state, if functional programming languages using lazy over strict evaluation tend to run faster, slower or equally as fast. 53.5% of the participants thought, that both evaluation strategies would be equally as fast. Given that lazy evaluation is unpredictable, this might also be interpreted as a "do not know". 34.9% believe, that lazy programs run faster and 11.6% think that they run slower as strictly evaluated implementations.

## 4.2. Concurrency & Parallelism

It is a widely spread opinion that functional programming languages are predestined for programming concurrent or parallel applications. In this section, it will become clear that this is only partly true.

### In Theory

Survey participants were confronted with the following statement: In theory, purely functional programs can be almost arbitrarily parallelised. In practice, it is therefore very easy to parallelise your programs written in a functional language. 69.8% thought this statement to be true. 11.6% felt that this statement if false. The other 18.6% believed that the statement is only partly true or that it depends largely on the programs under inspection.

Indeed it is true that at least theoretically, purely functional languages can be almost arbitrarily parallelized. The reason for this is referential transparency as was discussed briefly in Subsection 2.2.3. Referential transparency (and even more so laziness) abstracts over the execution order. This allows pure expressions that are used to construct pure functions, or in other words, functions that evaluate to the same result given the same input arguments. This means that each expression can be evaluated in parallel without influencing the computation result [20].

### In Practice

Fortunately, referential transparency also holds for functional programs in practice, meaning that programs can be parallelized without having to fear changes in the

programs behaviour. This is great. In imperative languages, one must always be aware of shared memory locations and prevent unintended behaviour by locking and releasing resources. Functional programming languages do not need such constructs. But, as usual, there is a catch. Hinsen wrote [35], an article in the promises on functional programming. In it, he states:

> *Although it's true that compilers for functional languages could in principle transform a serial into an equivalent parallel program automatically, there remains the problem of finding such a transformation that actually yields an efficient program for a given parallel computer and given input data. Compiler technology isn't yet up to this task, although this could well change in the future.*

Simply put, it is a non-solvable problem to automatically find a parallelized program transformation that actually runs faster than its serial counterpart. What sounded perfect in theory is put in question by thread creation overhead. An early experimental version of the Haskell Glasgow Compiler tried to make use of automatic program transformation. This resulted in programs spawning thread numbers in the hundreds, thousands and sometimes even millions of threads. Thread creation and context switching then completely dominated computation time [20].
There is some interesting research trying to solve the above problem. Sisal is an attempt on creating a parallel language, that supports implicit parallelization [15]. Unfortunately, the last version of SISAL was released in 2015. Runciman and Naylor published a paper about the Reduceron, a processor design for the only purpose of performing graph reduction [43]. They developed an own language called F-Lite, which is a subset of Haskell and Clean. The interesting part of their work is that the reduction is performed on self-reconfiguring FPGAs. Another approach to improve parallel behaviour of functional programs is the use of runtime profiling tools. In [38], the tool ThreadScope is discussed which is a browser on a graphical timeline enabling the developer to analyze his programs runtime behaviour. It will be interesting to see what the future brings regarding this topic.
Finally, it has to be discussed how the speedup of parallelized functional programs compares to imperative ones.

58.1% of survey participants think that the speedup is about the same. 34.9% estimate the gain to be bigger and 7% guess it to be smaller than for imperative programs.

For that purpose, Haskell has been chosen to represent the functional programming languages. The results shown in Table 4.1 stem from [5], a website whose purpose is the comparison of programming languages omptimized to the maximum. Note that

usually, such drastic optimizations are only needed in real-time critical applications which only represent a small fraction of actual applications in the real world.

| **Languages** | **Best Result** | **Worst Result** |
|---|---|---|
| Haskell vs. C++ | 1.61 x slower (fannkuch-redux) | 11.14 x slower (fasta) |
| Haskell vs. C | 1.86 x slower (fannkuch-redux) | 11.04 x slower (binary-trees) |
| Haskell vs. Java | 1.06 x faster (spectral-norm) | 6.89 x slower (fasta) |

Table 4.1.: Language benchmarks comparison

In addition to the bad results in time comparison, the highly optimized Haskell code does lack the usual conciseness functional programming languages are famous for.
The results described in this section can be summarised as follows:

- Functional programs can be parallelized without fear of destroying the programs correctness.
- An increase in performance is as hard to achieve using functional languages than it is using imperative ones.
- When looking at high performance computing, imperative programming languages dominate functional languages in almost every case.

These results directly compete the common opinion on functional programming and parallelism.

## 4.3. Community Support

Functional programming is often said to be lacking the critical mass to be seen as relevant. Section 3.3 of this thesis showed, that statistically speaking, functional programming languages are not widely spread, which supports the statement.
The results presented in this subsection stem from an interview conducted with Cyrill Schenkel, a fellow student of the author. Schenkel wrote a thesis on modern Integrated Development Environment (IDE) support for functional programming [48]. He developed an IDE for the language Haskell based on Visual Studio Code[1] and to gather the necessary requirements for his IDE, he did research using help of the Haskell community. By doing this, he got a good insight of how the Haskell community is organized. The interview transcript can be found in appendix C. The results are therefore only verified for the Haskell developer community. However, it is the authors opinion that for other functional programming languages, the community looks somewhat similar.

[1] `https://code.visualstudio.com`

The survey revealed that 48.8% of participants do not believe that functional programming is lacking the critical mass. 39.5% think the opposite. 11.6% elaborated further with interesting comments. One participant stated that it varies from industry to industry, but that it's seeing growth all over. React and similar projects are even bringing it to web development. In data pipeline engineering, it seems to be more the rule than the exception. Another one wrote: "What we lack is back education in fundamentals. As hand-wavy example: The courses that touch Turing machines are too few and the ones that dare talking lambda calculus even less."

The most important contact points of the Haskell community are the following:

- An Internet Relay Chat (IRC) channel that is very active[2]. Over 10'000 people are ready to answer questions. Usually, one can expect an answer to their question within five minutes.
- For simpler questions, there is an IRC beginners channel.
- There is a Reddit[3] Haskell group that is very active.
- In the region, there is the Zürich user group called HaskellerZ[4] .
- The Commercial Haskell group, which is a group composed of companies opened a GitHub organization[5], where modified preludes and other useful contributions can be found.

With 10'000 active users on the main resource for answering questions the community seems rather small. However, the rapid response time seems quite unique, even when compared to a lot of popular languages. The community is therefore not seen as problematic regarding the adoption of functional programming languages by the author.

Survey participants see a larger problem in the education provided in functional programming. 69.8% think that there is not enough done. 20.9% do think the opposite. 9.3% elaborated that there needs to be a much greater focus on what makes functional programming a style rather than a language characteristic. The quality needs to go up, not the quantity and that there is a lot of good material out there, but perhaps not enough work done at teaching FP at universities.

[2] `https://www.haskell.org/irc`
[3] `https://www.reddit.com`
[4] `https://www.meetup.com/HaskellerZ`
[5] `https://github.com/commercialhaskell/commercialhaskell`

## 4.4. Library & IDE Support

One conclusion drawn from Section 3.2 was the importance of tooling and library support to increase language adoption probability. This section discusses the library and IDE support for functional programming languages, once again using Haskell as the representative. As in Section 4.3, the results were gathered through the interview with Schenkel, who got great insight in what is missing regarding tooling in Haskell.

Participants of the survey were asked to categorize the tooling support for functional programming languages. The results are as follows: 11.6% voted for excellent tooling support, 25.6% thought it to be good, 25.6% ok, 23.3% not so good and 14% even thought it was very bad.

### 4.4.1. Libraries

Haskell comes with the so called standard prelude, a collection of standard packages. The prelude includes the most common libraries like `Data.List.Utils`, `Data.String.Utils`, `Data.Dates`, `System.IO`, `Network.Socket` and many more. Third party libraries like `Crypto-API`, `Crypto-Cipher`, `RSA-Haskell`, `HDBC` and `HSQL` provide the missing pieces to make Haskell ready for practice. A list of recommended Haskell libraries can be found at [3]. This does sound promising. But how is the quality of the provided prelude and third party libraries? Schenkel came to the following conclusions:

- There is no platform independent User Interface (UI) library that is usable.
- Standard strings are not to be used (no good logging libraries do).
- The prelude has historically grown and was never broken to maintain backward compatibility.
- Naming conventions for libraries were introduced in a later point in time, so there are old libraries that do not conform the conventions.
- Almost all companies define their own Prelude, because the standard prelude is not good enough.
- A lot of third party libraries are useless, because everyone can upload their package to Hackage[6]. Sometimes this is even used for personal backups.
- As a result, there are a lot of unmaintained libraries.

[6] `https://hackage.haskell.org`

The conclusion after summarising all these critical issues is that library support definitely is a problem. A quick check revealed that for Standard ML, the situation looks quite similar [25]. So far, this is the biggest issue regarding functional programming adoption.

### 4.4.2. IDE Support

There are a lot of attempts to develop IDEs for Haskell. Interestingly, the definition if IDE seems to drift apart from what developers of imperative languages understand under the term. Cabal, Stack, Intero, EclipseFP, Leksah, Haskell for Mac, GHCMod, GHCIDE, Hare and HIE are the most known such development environments. However, most of them are just dependency managers and build tools. Some of them offer, autocompletion and show type errors in addition to provide syntax highlighting. What is missing in almost all of the above mentioned tools is the ability to do automatic refactoring. One might argue that for functional languages, refactorings are not necessary. And it is true, that a lot of refactorings do not make sense on functional programming languages. But there are some that would be useful, such as formatting, function extraction and renaming [49].
In the author's opinion, a well equipped IDE can be of great help when learning a new programming language. Using glorified text editor plugins and a command line tool running the compiler seems to be an approach better suited for advanced programmers. IDE support can therefore be seen as an additional issue regarding language adoption.

# 5. Application Areas

Finding undiscovered application areas for functional programming languages is not an easy task. Especially since experts in any kind of programming style are well aware of the advantages and disadvantages of said style. This, of course, results in early birds exploring the so far undiscovered terrain. To be able to identify such an application area, a tactic had to be developed during the course of this thesis. The following sections describe the steps taken in chronological order.

## 5.1. Dividing the Area of Applications

For identifying areas where functional programming is not used, a coarse division of the area of applications has been undertaken. The resulting application area matrix can be seen in Figure 5.1.

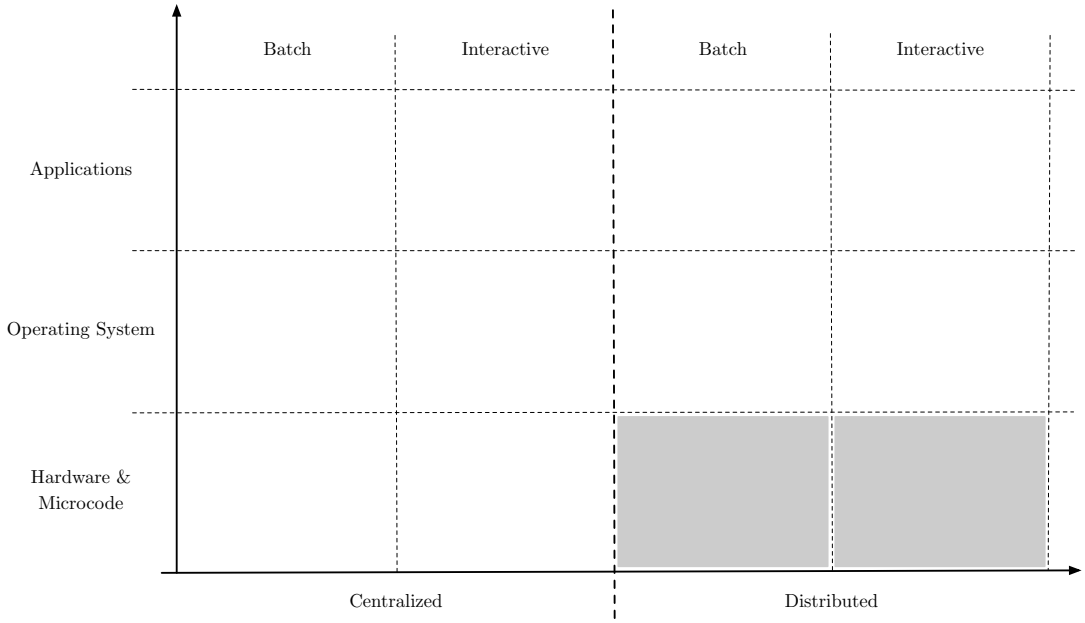


Figure 5.1.: Application areas matrix skeleton

Horizontally, the domain of application areas has been divided into centralized and

distributed applications. Each of these parts has been further divided into batch and interactive applications. Vertically, three categories were introduced, namely hardware & microcode, operating system and applications. What stands out is the grey areas in the hardware & microcode category on the distributed side of the matrix. After careful consideration, the author decided that these areas can never be inhabited, because distribution is handled earliest on the operating system level. Of course there might be hardware specialized in distributed computing, but, because the interest lies on functional programming languages, such applications cannot occur. Note that not all kinds of applications can be sorted into this matrix. For example, there exist lots and lots of interactive applications that have some sort of batch computation logic in them.

## 5.2. Defining Non-Functional Requirements

After defining application categories, the next step was to assign Non-Functional Requirements (NFRs) to each cell in the application area matrix. Figure 5.2 shows the result.

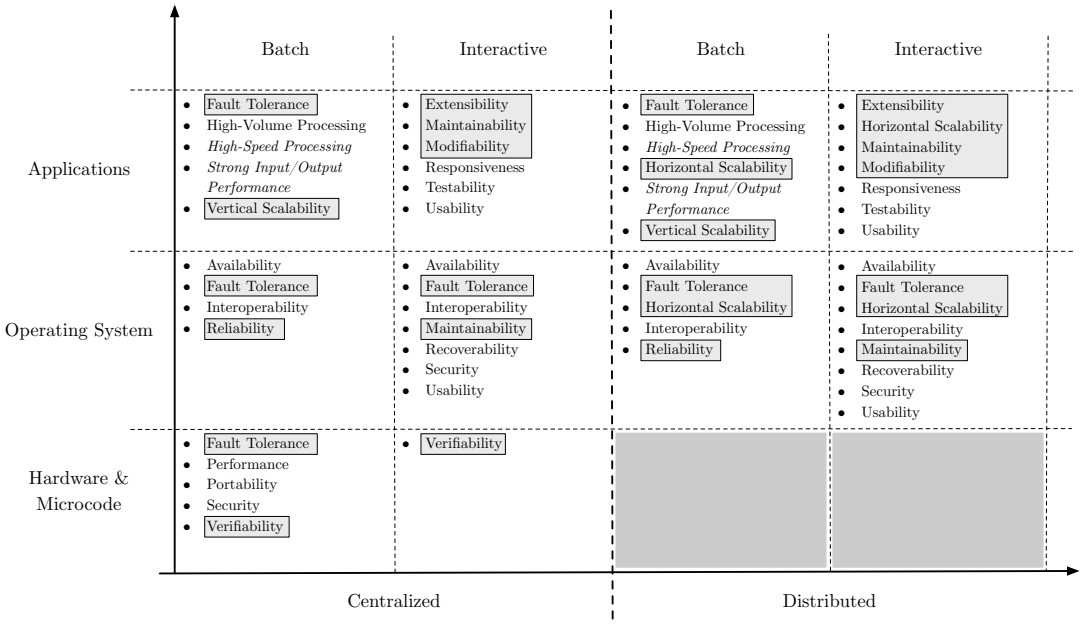


Figure 5.2.: Application areas matrix requirements

The NFRs well suiting for the functional programming paradigm are highlighted in grey boxes. NFRs that do not match well with functional programming are highlighted in *italic* font. What sticks out is that every cell has requirements that

suit functional programming. The only cells with requirements against functional programming are batch applications, centralized as well as distributed. Batch applications often require high-speed processing and strong input/output performance (but not always). As was established in Sections 4.1 and 4.2, there are better language options for high-performance applications. Besides that, functional programming languages can be used in every other application area. However, it will be shown in Section 5.3, that one of the most wide spread use cases for functional programming languages falls in said category.

## 5.3. Search for Existing Applications

The next step taken was to search for application areas where functional programming was already used. This included not only current and active applications, but also no longer existing or discontinued ones. This makes sense because it tells something about the usability of the paradigm for the given application area whether it was used often or only once and whether it is still used today.

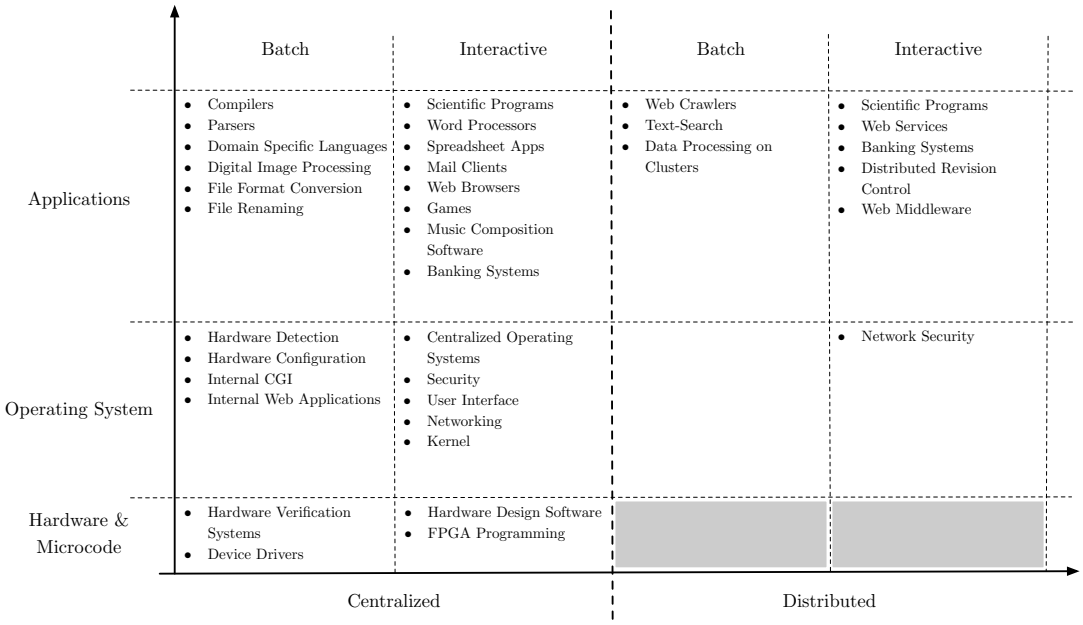


Figure 5.3.: Application types

Figure 5.3 lists application areas where functional programming has been used. Note, that the author does not claim completeness. Each of these application types has at least one implementation using the functional programming paradigm. The most known and established use cases can be found in two areas. The first use

case is the definition of domain specific languages and corresponding parsers or compilers. This is the one use case functional programming is predestined for, because of its pattern matching ability. Note that despite Section 5.2 marked this category as ill suited, high-speed processing and and strong input and output performance are not strong requirements for this use case. The second use case concerns scientific programs. It does not come as a surprise that programs relying on mathematics can be implemented well in functional programming languages. Mathematica and R are good examples for functional programming languages used for scientific applications.
Listings of applications in Haskell can be found at [2, 11, 12] and [46]. An enumeration of success stories of companies using the OCaml programming language can be found at [4].
One application area shall be pointed out. There are a few projects where functional programming is used to design and verify hardware. For example, Lava is a tool to assist circuit designers in specifying, designing, verifying and implementing hardware [30]. Lava was a subset of Haskell and is called the Bluespec[1] programming language today. Another example is the language reFLect, a functional programming language close to ML with reflection features intended for applications in hardware design and verification [32]. Functional programming is a good fit for this kind of applications because it abstracts away the actual hardware design. Instead, the developer can define only the hardware's intended functionality and leave the synthesis of the hardware to a compiler.
Overall it can be stated that a lot of application areas have been tackled using functional programming. Some more and some less successful. Huge success stories are definitely missing, not only for Haskell, but for all functional programming languages. A closer look on success story wishes and promising application areas is provided in Section 5.4. Closing this section, a statement from [21] is given below, summarising the findings so far:

> *Practically everything you can do in procedural programming can be done in functional programming, and the same thing in reverse. It's just another way to code something – another perspective on the problem and a different way to solve it. However, because not many people use functional programming, the problem is more about the lack of good libraries, portability/maintainability (since the maintainer has a better chance to understand something written in C++ than Scheme), and the lack of documentation and community.*

[1] `http://bluespec.com`

## 5.4. Identify Promising Areas

The final question of the survey conducted during this thesis was not only answered by all participants, but revealed a very clear result. 95.3% of survey participants think that functional programming languages could be more often used in practice (industry/enterprise projects). Only 4.7% beg to differ.

The previous sections of this chapter provided background information on the current state of functional programming regarding application areas. It was shown how the realm of applications can be divided into categories and what the non-functional requirements for these categories are. Then, a quick overview of already covered application ares was given. The aim of this section is to take this information and identify promising application areas for the functional programming paradigm. But first, the main advantages of functional programming are listed below, because this was somewhat neglected so far.

- **Conciseness**: Code written in functional languages is often very concise due to the high abstraction level.
- **Abstraction**: The powerful abstraction mechanisms of functional programming help to deal with complexity.
- **Modularity**: Functional code bases provide a high level of modularity, because every function defined can be reused (if exported).
- **Reusability**: A direct consequence of the modularity of functional code is reusability.
- **Correctness**: The strong type systems of a lot of functional programming languages ensure correctness of the written code, based on mathematical properties.
- **Domain Specific Languages**: Although more of a use case, the ability of functional programming languages to specify and parse domain specific languages using pattern matching can be listed as a definitive advantage.

The following subsections each discuss an application area that seems promising to the author in terms of functional programming adaptation.

### 5.4.1. Web Applications

Web applications have grown in importance in the past few years. A lot of devices are capable of connecting to the internet and communicate to web Application

Programming Interfaces (APIs). Although functional programming is sometimes used to write web applications, the occurrences are few. But why should one use functional programming for web applications? Firstly, the Hypertext Transfer Protocol (HTTP) is stateless. If some form of context shall be preserved, it has to be encoded somehow in the requests and responses. This suits functional programming well, because no state has to be observed, and there are no side effects. Secondly, pattern matching can be used to differentiate incoming requests and then call the corresponding function. Thirdly, incoming requests can easily be processed concurrently. Note that this does not necessarily denote an advantage to other programming paradigms. Lastly, most functional programming languages allow to recompile only partial programs in a running environment, which simplifies software maintenance [8].
What is missing for most functional programming languages is a solid and easy to use web framework, on top of which the web applications can be programmed. Of course, the number and quality of web frameworks has increased in the last few years. But that these frameworks have not reached the state of the art yet can be seen at [1], where Haskell users state their wishes for success stories of Haskell in the industry. Successful web applications are a recurring subject in this wish list. Therefore, success stories of functional web applications could be a huge boost in functional programming adoption.

### 5.4.2. Hardware Programming & Design

In Section 5.3, some applications of functional programming for hardware programming and design have been quickly discussed. One of which was the Bluespec programming language, a subset of Haskell which is used as a high-level hardware description language. Once the design is complete, the Bluespec compiler generates synthesizable Verilog. This apporach has three impactful advantages. Firstly, using a functional language allows to design the hardware on a high abstraction level. It is defined what the hardware is supposed to do instead of defining every circuit. This also enables programmers with only little knowledge about hardware to design their own circuits. Secondly, hardware designs generated from functional code turn out to be performing better than corresponding ones design using Verilog or VHDL [7]. Lastly, the mathematical properties of programs written in functional languages allow different kinds of verification. In addition, the strong type systems ensure program correctness.
The reason why this seems to be a promising application area is the growing use of embedded hardware on more and more devices. Also, a lot of systems like Supervisory Control And Data Acquisition (SCADA) systems make use of Programmable Logic Controllers (PLCs). Today, there is a wide variety of ways such a controller can be programmed. Usually, this is depending on the manufacturer

of the device. An additional abstraction layer on top of these approaches using functional programming (like Bluespec on Verilog) would make the programming easier, uniform and more accessible to non-professional hardware designers.

### 5.4.3. Enterprise Resource Planning Systems

Enterprise Resource Planning (ERP) systems are a part of every major company. These systems usually combine three layers. Interface services form the presentation layer and database access services form the data access layer. The third layer called business layer comprises all the business logic services. These business logic services could be programmed using a functional approach. The business logic usually involves the processing of large data sets, for which functional programming is well-suited, again because of its vertical scalability characteristic.
A successful use of functional programming in an ERP system could have a large impact on functional programming adoption. At [1], the wish for success stories in this area was also widely spread.

### 5.4.4. Mission Critical Systems

There are some application areas where the correctness of applications is mission critical, in the sense that a failure has a large impact on security or financial loss. Examples of such systems are air traffic control systems, aerospace systems, pace makers and many more. Most of such systems are written in languages that have verification included in the language itself. Ada is such a language and Spark is its most common dialect.
It would be interesting to see if functional programming languages could be used in the place of Ada to implement mission critical systems. A static type system ensures program correctness and a separate verification using the mathematical properties of functional programs could be used to implement a semantic verification. However, if hard real-time responses are a requirement, functional programming languages are considered to be a bad choice in the opinion of the author. For this case, languages that allow manual memory management should be preferred [23].

### 5.4.5. Interesting Research Areas

A lot of research papers referenced in this thesis determine that the currently used computer architecture based on the von Neumann architecture are not particularly well-suited for running functional programs, the most prominent being [28]. During the period this thesis was conducted, two older publications came to the authors attention, which might be the basis for interesting research projects. In [50], Trelean evaluates systematically, what it would take to design a computer architecture

fitting the functional programming paradigm. Building such a system would be both challenging interesting. In [33], Henderson describes a purely functional operating system. Having both a computer architecture suited especially for functional programming and an operating system on top of it, would reveal if the von Neumann architecture really is a problem regarding functional program performance.

# 6. Results

In this chapter, the results of the thesis are summarized and evaluated according to the goals stated in Section 1.2.
First of all, the survey conducted during the period of the thesis has shown, that there are a lot of different opinions regarding functional programming. The often very even distribution of contradicting answers on almost every survey question leads to the conclusion, that a lot of these opinions are in fact speculations and ill-founded. The problem described in Section 1.1 can therefore be considered to be valid.
In Section 5.3, application areas where functional programming is or was used were identified and categorized and sources for more detailed information were listed. Because investigating every application area separately would have taken a lot longer than the period of this thesis, the adoption of functional programming languages in numbers was analyzed in general in Section 3.3. This approach made sense because it turned out that language adoption mostly depends on the availability of open source projects, libraries and frameworks. Language features have only limited influence on adoption. Chapter 4 discussed general opinions of functional programming and verified their validity. It was shown that functional programs do not perform as bad as is the most widely spread opinion, but also, that writing parallel applications is not as much of an advantage as promised. Furthermore, it was shown that the community support for functional programming languages is quite healthy, but that the tooling for development is not state of the art compared to other programming languages. Advantages of the functional programming paradigm were quickly discussed in Section 5.4.
Although not linked directly to specific applications areas, Chapter 4 proposed and investigated the reasons why the wide use of functional programming has been limited.
Section 5.4 finally proposed promising application areas for which functional programming seems a great fit. These are web applications, hardware programming and design, enterprise resource planning systems and mission critical systems. Not only would functional programming fit these application areas, but according to community surveys, success stories in these areas would be very welcome. In Chapter 7, measures that have to be taken to take functional programming further are described.

# 7. Conclusion & Future Work

The research conducted for this thesis revealed several problems that need to be solved in order to increase the chances of bettering the functional language adoption in the industry. The results were gathered by first looking at language adoption in general and then analyzing functional programming according to the findings. In doing that, some of the myths and ill-founded opinions about the functional programming paradigm were dissolved. Finally, the realm of application areas was categorized, existing functional applications were analyzed and promising application areas were identified. What is left to be done is to list the specific problems to be solved for bettering the adoption of functional programming. This is done in the following:

- **Missing Success Stories**: A key factor of successful language adoption are open source projects and well programmed libraries and frameworks. In other words, publicly available success stories lead to better language adoption. In order to increase the adoption of functional programming, successful projects should be open sourced.
- **Tooling Support**: The existing tooling for functional programming languages is not state of the art. An IDE with syntax highlighting, dependency management and some form of refactorings should not be too much to ask for. Improved tooling would definitely have an impact on adoption.
- **Education**: The education in functional programming often comprises of one single course that can be optionally visited at universities. Functional programming should be presented as an equivalent alternative to object oriented programming in the syllabus of every computer science student.

## 7.1. Future Work

The time period of this thesis was very limited regarding the size of the task. In the following, possible future work to improve the results of this thesis are listed:

- The empirical analysis on language adoption was conducted independent of the programming paradigm. A closer study only including functional

programming languages might reveal adoption criteria especially important for functional programming languages.

- The historical data for the TIOBE index is not openly accessible. One has to buy the complete data sets. A different popularity index with an open history could reveal the development of the popularity of functional programming languages.
- Language specific features were often only researched for Haskell and in some cases Standard ML because of the limited time frame of the thesis. The same research could be extended for other functional programming languages.
- Specific application areas could be looked at in more detail. Such a study would probably require a lot of effort and ideally a team of researchers.

# Bibliography


[1] 72 would-be commercial haskell users: what haskell success stories we need to see. https://www.reddit.com/r/haskell/comments/377zyc/72_wouldbe_commercial_haskell_users_what_haskell. Accessed: 2017-07-26.

[2] Applications and libraries. https://wiki.haskell.org/Applications_and_libraries. Accessed: 2017-07-24.

[3] Awesome haskell. https://github.com/krispo/awesome-haskell. Accessed: 2017-07-18.

[4] Companies using ocaml. https://ocaml.org/learn/companies.html. Accessed: 2017-07-24.

[5] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org. Accessed: 2017-07-15.

[6] Disadvantages of purely functional programming. http://flyingfrogblog.blogspot.ch/2016/05/disadvantages-of-purely-functional.html. Accessed: 2017-07-13.

[7] Erlang factory sf 2016 - keynote - john hughes - why functional programming matters. https://www.youtube.com/watch?v=Z35Tt87pIpg. Accessed: 2017-07-26.

[8] Functional programming for the rest of us. http://www.defmacro.org/ramblings/fp.html. Accessed: 2017-07-13.

[9] Functional programming in industry. http://flyingfrogblog.blogspot.ch/2007/09/functional-programming-in-industry.html. Accessed: 2017-07-13.

[10] Ghc/memory management. https://wiki.haskell.org/GHC/Memory_Management. Accessed: 2017-07-14.

[11] Haskell in industry. https://wiki.haskell.org/Haskell_in_industry. Accessed: 2017-07-24.

[12] Hssuccess.md. https://gist.github.com/ekalinin/04a538f2918b4685fcfd65982e471ee9. Accessed: 2017-07-24.

[13] Is functional programming overtaking the it industry? https://hackernoon.com/is-functional-programming-overtaking-the-it-industry-c0c5a535818a. Accessed: 2017-07-13.

[14] Pitfalls/disadvantages of functional programming. https://stackoverflow.com/questions/1786969/pitfalls-disadvantages-of-functional-programming. Accessed: 2017-07-13.

[15] Sisal parallel programming. https://sourceforge.net/projects/sisal. Accessed: 2017-07-15.

[16] Tiobe index for july 2017. https://www.tiobe.com/tiobe-index. Accessed: 2017-07-12.

[17] Tiobe programming community index definition. https://www.tiobe.com/tiobe-index/programming-languages-definition. Accessed: 2017-07-12.

[18] What are some limitations/disadvantages of functional programming? https://www.quora.com/What-are-some-limitations-disadvantages-of-functional-programming. Accessed: 2017-07-13.

[19] What do haskellers want? over a thousand tell us. https://www.fpcomplete.com/blog/2015/05/thousand-user-haskell-survey. Accessed: 2017-07-13.

[20] What is it about functional programming that makes it inherently adapted to parallel execution? https://softwareengineering.stackexchange.com/questions/293851/what-is-it-about-functional-programming-that-makes-it-inherently-adapted-to-para. Accessed: 2017-07-15.

[21] When to use a functional programming language? https://stackoverflow.com/questions/397425/when-to-use-a-functional-programming-language. Accessed: 2017-07-24.

[22] When to use functional programming languages and techniques. http://www.techrepublic.com/blog/software-engineer/when-to-use-functional-programming-languages-and-techniques. Accessed: 2017-07-13.

[23] Which languages are used for safety-critical software? https://stackoverflow.com/questions/243387/which-languages-are-used-for-safety-critical-software. Accessed: 2017-07-26.

[24] Why hasn't functional programming taken over yet? https://stackoverflow.com/questions/2835801/why-hasnt-functional-programming-taken-over-yet. Accessed: 2017-07-13.

[25] Wish list for sml. https://github.com/standardml/hackday/wiki/Wish-list-for-SML. Accessed: 2017-07-18.

[26] S. Adams. Implementing sets efficiently in a functional language. 1992.

[27] S. Adams. Functional pearls efficient sets—a balancing act. *Journal of functional programming*, 3(4):553–561, 1993.

[28] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[29] H. Barendregt. Functional programming and lambda calculus. In *Handbook of theoretical computer science (vol. B)*, pages 321–363. MIT Press, 1991.

[30] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[31] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[32] J. Grundy, T. Melham, and J. O'leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

[33] P. Henderson. *Purely functional operating systems*. Functional programming and its applications, Cambridge University Press, 1982.

[34] P. Henderson. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering*, (2):241–250, 1986.

[35] K. Hinsen. The promises of functional programming. *Computing in Science & Engineering*, 11(4), 2009.

[36] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[37] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[38] D. Jones Jr, S. Marlow, and S. Singh. Parallel performance tuning for haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 81–92. ACM, 2009.

[39] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

[40] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[41] L. A. Meyerovich and A. S. Rabkin. Socio-plt: Principles for programming language adoption. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 39–54. ACM, 2012.

[42] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, 48(10):1–18, 2013.

[43] M. Naylor and C. Runciman. The reduceron reconfigured. In *ACM Sigplan Notices*, volume 45, pages 75–86. ACM, 2010.

[44] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[45] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[46] B. O'Sullivan, J. Goerzen, and D. B. Stewart. *Real world haskell: Code you can believe in.* " O'Reilly Media, Inc.", 2008.

[47] C. G. Ponder, P. McGeer, and A. P. Ng. Are applicative languages inefficient? *ACM SIGPLAN Notices*, 23(6):135–139, 1988.

[48] C. Schenkel and F. Bitterlin. Modern ide support for functional programming. 2017.

[49] S. Thompson, C. Reinke, et al. Refactoring functional programs. *Advanced Functional Programming*, 3622:331–357, 2004.

[50] P. C. Treleaven. *Computer architecture for functional programming*. Cambridge Univ. Press, 1982.

[51] D. Turner. Some history of functional programming languages. In *International Symposium on Trends in Functional Programming*, pages 1–20. Springer, 2012.

[52] P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

# Appendix

# A. GitHub Repository Data

Table A.1 shows the data gathered using the GitHub **API!** (**API!**).

| | **C** | **Scheme** | **Java** | **Scala** | **Python** | **JavaScript** | **C++** | **C#** |
|---|---|---|---|---|---|---|---|---|
| **13Q1** | 118312 | 796 | 197745 | 17794 | 246654 | 295702 | 113914 | 36286 |
| **13Q2** | 129389 | 928 | 207578 | 18193 | 284710 | 335580 | 123877 | 42984 |
| **13Q3** | 146011 | 1655 | 264444 | 23081 | 355137 | 453975 | 153883 | 59691 |
| **13Q4** | 174106 | 1540 | 347554 | 27393 | 422065 | 574970 | 209157 | 78711 |
| **14Q1** | 217608 | 2111 | 442589 | 36583 | 535857 | 769264 | 262408 | 94680 |
| **14Q2** | 229411 | 1501 | 504917 | 40503 | 595429 | 905960 | 291712 | 116265 |
| **14Q3** | 246508 | 2317 | 534936 | 49419 | 655246 | 973300 | 319109 | 134294 |
| **14Q4** | 288655 | 2144 | 602979 | 54763 | 714492 | 1106165 | 402005 | 155859 |
| **15Q1** | 335870 | 2519 | 773337 | 66602 | 912399 | 1398294 | 437501 | 231320 |
| **15Q2** | 380079 | 3725 | 890923 | 77851 | 1036089 | 1594011 | 485118 | 270716 |
| **15Q3** | 400426 | 2771 | 934734 | 85424 | 1156497 | 1855388 | 540097 | 297000 |
| **15Q4** | 428380 | 2623 | 1030562 | 84626 | 1232022 | 2034584 | 588587 | 352323 |
| **16Q1** | 487917 | 4252 | 1130939 | 106247 | 1340496 | 2327114 | 652851 | 411419 |
| **16Q2** | 468270 | 4696 | 1140974 | 99358 | 1362310 | 2356473 | 647318 | 390957 |
| **16Q3** | 412837 | 4827 | 1095251 | 95731 | 1279652 | 2128212 | 719004 | 367238 |
| **16Q4** | 441545 | 4256 | 1125449 | 102009 | 1305253 | 2237569 | 873094 | 382828 |
| **17Q1** | 389108 | 4087 | 1021431 | 99783 | 1285800 | 2009678 | 639028 | 412335 |
| **17Q2** | 406380 | 5254 | 1257408 | 121259 | 1530641 | 2405032 | 717768 | 625156 |
| | **F#** | **Haskell** | **Lisp** | **R** | **Erlang** | **Swift** | **Matlab** | |
| **13Q1** | 1045 | 19424 | 2410 | 1739 | 6350 | 231 | 2457 | |
| **13Q2** | 1433 | 18242 | 2917 | 1617 | 6994 | 729 | 3374 | |
| **13Q3** | 1880 | 23234 | 2680 | 4112 | 7593 | 450 | 3655 | |
| **13Q4** | 3580 | 24439 | 3223 | 6460 | 9858 | 1421 | 5099 | |
| **14Q1** | 4531 | 31970 | 3764 | 9129 | 10882 | 1732 | 6680 | |
| **14Q2** | 5932 | 47230 | 3937 | 13889 | 12203 | 6662 | 7753 | |
| **14Q3** | 7422 | 42279 | 4000 | 14630 | 11212 | 16600 | 8334 | |
| **14Q4** | 8919 | 45246 | 5425 | 15996 | 13552 | 23338 | 10505 | |
| **15Q1** | 11228 | 56442 | 6119 | 23235 | 14995 | 37184 | 14614 | |
| **15Q2** | 10127 | 59889 | 6098 | 29983 | 18423 | 53491 | 17222 | |
| **15Q3** | 11928 | 65627 | 5949 | 34711 | 22169 | 75132 | 17241 | |
| **15Q4** | 14196 | 65633 | 5551 | 33678 | 21296 | 94430 | 17367 | |
| **16Q1** | 13671 | 64400 | 5800 | 38204 | 20755 | 120949 | 20388 | |
| **16Q2** | 13234 | 65029 | 5532 | 39146 | 19992 | 123782 | 22667 | |
| **16Q3** | 9711 | 54824 | 5275 | 37164 | 22359 | 127694 | 18696 | |
| **16Q4** | 9800 | 52652 | 5514 | 36596 | 17675 | 133515 | 20012 | |
| **17Q1** | 9027 | 52104 | 5168 | 35556 | 17575 | 105842 | 17967 | |
| **17Q2** | 9487 | 55327 | 5835 | 48257 | 15749 | 125148 | 24936 | |

Table A.1.: GitHub repository data

# B. Survey Data

This chapter lists the raw data gathered from a survey conducted by the author during the time period of this thesis. The data had to be separated into different tables because of the limited space of the document.

| Id | Age | Profession | How did you learn about the existence of the functional programming paradigm? (multiple answers allowed) |
|---|---|---|---|
| 1 | 37 | Lecturer / Engineer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends);Industry (as part of your job) |
| 2 | 30 | Student | Lecture at university (or course of some sort at a school);Industry (as part of your job);Other... |
| 3 | 25 | Software Engineer | Lecture at university (or course of some sort at a school) |
| 4 | 28 | Project Staff at IFS | Other... |
| 5 | 59 | Lecturer | Self-taught (online courses or friends) |
| 6 | 22 | CS Student @ HSR | Self-taught (online courses or friends) |
| 7 | | Prof | Other... |
| 8 | | Software Engineer | Self-taught (online courses or friends) |
| 9 | 34 | Software Engineer | Lecture at university (or course of some sort at a school) |
| 10 | 29 | Software Developer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 11 | | Software Developer | Other... |
| 12 | | Software Developer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 13 | | Student | Lecture at university (or course of some sort at a school) |
| 14 | 27 | Software Developer | Self-taught (online courses or friends) |
| 15 | 34 | Software Engineer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends);Industry (as part of your job);Other... |
| 16 | 28 | Computer Scientist | Lecture at university (or course of some sort at a school) |
| 17 | 24 | Game Programmer | Self-taught (online courses or friends) |
| 18 | | Software Engineer | Self-taught (online courses or friends) |
| 19 | 32 | Data Engineer | Other... |
| 20 | | Software Engineer | Self-taught (online courses or friends) |
| 21 | 25 | Software Engineer | Self-taught (online courses or friends) |
| 22 | | Software engineer | Lecture at university (or course of some sort at a school) |
| 23 | 26 | EE | Lecture at university (or course of some sort at a school) |
| 24 | 31 | Senior Software Developer | Self-taught (online courses or friends) |
| 25 | 39 | Computer Programmer | Other... |
| 26 | 24 | Software Engineer | Self-taught (online courses or friends);Other... |
| 27 | 27 | Software Engineer | Self-taught (online courses or friends) |
| 28 | 33 | Computer scientist | Self-taught (online courses or friends);Other... |
| 29 | | Software Engineer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 30 | | Student | Self-taught (online courses or friends) |
| 31 | | Intern Electronic Engineer | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 32 | 23 | Functional data scientist | Self-taught (online courses or friends) |
| 33 | 31 | Developer | Self-taught (online courses or friends) |
| 34 | 28 | Programmer | Self-taught (online courses or friends) |
| 35 | | Student | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 36 | 62 | Programmer | Self-taught (online courses or friends) |
| 37 | 25 | Software Engineer | Lecture at university (or course of some sort at a school) |
| 38 | 32 | Senior Software Engineer | Lecture at university (or course of some sort at a school) |
| 39 | 20 | Student Computer Science | Lecture at university (or course of some sort at a school) |
| 40 | 36 | Haskell Programmer | Lecture at university (or course of some sort at a school) |
| 41 | | Professor | Lecture at university (or course of some sort at a school);Self-taught (online courses or friends) |
| 42 | 47 | Physicist and other things | Self-taught (online courses or friends) |
| 43 | 60 | Quantitative Analyst / Developer | Self-taught (online courses or friends) |

Table B.1.: Career & education part 1

| Id | If you selected other, please state your custom answer. | How often do you use functional programming languages or mixed paradigm languages with functional features? |
|---|---|---|
| 1 | | Weekly |
| 2 | As part of multiple thesis conducted for the master studies. | Daily |
| 3 | | Daily |
| 4 | Talk at the Cosin 2014 | Daily |
| 5 | | Daily |
| 6 | | Daily |
| 7 | Acquired it over the years | Weekly |
| 8 | | Daily |
| 9 | | Daily |
| 10 | | Daily |
| 11 | Workplace back in 2010 (before it was cool) | Weekly |
| 12 | | Daily |
| 13 | | Weekly |
| 14 | | Daily |
| 15 | Many conferences (C++Now, MeetingC++), talking to Phil Nash | Daily |
| 16 | | Daily |
| 17 | | Daily |
| 18 | | Weekly |
| 19 | High school computer science class | Daily |
| 20 | | Monthly |
| 21 | | Monthly |
| 22 | | Weekly |
| 23 | | A few times a year |
| 24 | | Weekly |
| 25 | I was searching for a way to make Java's reflection more convenient, stumbled upon Scala's structural typing. | Daily |
| 26 | #scalaz on freenode | Daily |
| 27 | | Daily |
| 28 | Open source projects, IRC | Daily |
| 29 | | A few times a year |
| 30 | | Daily |
| 31 | | Daily |
| 32 | | Daily |
| 33 | | Daily |
| 34 | | Weekly |
| 35 | | Monthly |
| 36 | | Daily |
| 37 | | Weekly |
| 38 | | Daily |
| 39 | | A few times a year |
| 40 | | Daily |
| 41 | | Daily |
| 42 | | Weekly |
| 43 | | Daily |

Table B.2.: Career & education part 2

| Id | In what context do you use functional programming? (multiple answers allowed) | If you selected other, please state your custom answer. | How long have you been using functional programming for? | What is the functional programming language of your choice? |
|---|---|---|---|---|
| 1 | Work;School;Personal projects | | >10 years | ML, OCaml, Haskell |
| 2 | Work;School | | 1 - 3 years | Haskell |
| 3 | Work;School;Personal projects | | 1 - 3 years | Python |
| 4 | School;Personal projects | | 3 - 5 years | Hakell |
| 5 | School;Personal projects | | >10 years | SML, Haskell |
| 6 | School;Personal projects | | 3 - 5 years | Haskell |
| 7 | Work;School;Personal projects | | >10 years | C++ |
| 8 | Work;Personal projects | | 5 - 10 years | Scala |
| 9 | Work;Personal projects | | 3 - 5 years | Java |
| 10 | Work;Personal projects | | 3 - 5 years | Scala |
| 11 | Work;Personal projects | | <1 year | Swift |
| 12 | Work | | 3 - 5 years | Scala |
| 13 | Work;School | | <1 year | Haskell |
| 14 | Work;School | | 1 - 3 years | JavaScript (mixed functional Programming) |
| 15 | Work;Personal projects | | 5 - 10 years | Pure functional is Haskell, mixed is C++ |
| 16 | Work;School;Personal projects | | 3 - 5 years | C# (i.a. LINQ) |
| 17 | Work;Personal projects | | 1 - 3 years | C++ |
| 18 | Work;Personal projects | | 3 - 5 years | scala |
| 19 | Work;Personal projects | | >10 years | Scala, but I use functional design in Python, C, Rust, etc also |
| 20 | Other... | C++ template metaprogramming | <1 year | Haskell |
| 21 | School | | 3 - 5 years | Scheme |
| 22 | School;Personal projects | | 1 - 3 years | Scala |
| 23 | School | | <1 year | Haskell |
| 24 | Personal projects | | 1 - 3 years | Haskell |
| 25 | Work;School;Personal projects;Other... | I would consider lambda calculus to teach logic, programming and possibly other subjects. | 5 - 10 years | Scala |
| 26 | Work;School;Personal projects | | 5 - 10 years | Haskell, Coq |
| 27 | Work;Personal projects | | 5 - 10 years | Haskell |
| 28 | Work;School;Personal projects | | 3 - 5 years | Scala, Idris, Haskell |
| 29 | Work;School | | 3 - 5 years | Scala |
| 30 | Work;School;Personal projects | | 1 - 3 years | Haskell |
| 31 | School;Personal projects | | 1 - 3 years | Idris |
| 32 | Work;School;Personal projects | | 3 - 5 years | Haskell |
| 33 | Work;Personal projects | | 3 - 5 years | Haskell |
| 34 | Work;Personal projects | | >10 years | Haskell |
| 35 | School;Personal projects | | 5 - 10 years | Haskell |
| 36 | Work;Personal projects | | 3 - 5 years | Haskell |
| 37 | Work;Personal projects | | 3 - 5 years | Haskell |
| 38 | Work;Personal projects | | 5 - 10 years | Haskell |
| 39 | School | | <1 year | Haskell |
| 40 | Work;Personal projects | | >10 years | Haskell |
| 41 | Work | | >10 years | Haskell |
| 42 | Work;Personal projects | | 1 - 3 years | Haskell |
| 43 | Work;Personal projects | | >10 years | Haskell |

Table B.3.: Career & education part 3

| Id | Programs written in functional programming languages tend to run _____ corresponding imperative implementations. | Functional programming languages using lazy over strict evaluation tend to run _____ . | Immutable data structure operations tend to run _____ corresponding imperative implementations. | Programs written in functional programming languages tend to allocate _____ memory than/as corresponding imperative implementations. | Immutable data structures use up _____ memory than/as corresponding imperative implementations. | Do you have comments on the topic performance that have not been covered in this section? |
|---|---|---|---|---|---|---|
| 1 | ...slower than ... | ...faster... | ...equally as fast as... | ...more... | ...less... | |
| 2 | ...slower than ... | ...faster... | ...slower than... | ...more... | ...more... | |
| 3 | ...slower than ... | ...faster... | ...faster than... | ...more... | ...more... | |
| 4 | ...equally as fast as... | ...faster... | ...slower than... | ...more... | ...less... | |
| 5 | ...slower than ... | ...slower... | ...slower than... | ...more... | ...more... | |
| 6 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...more... | ...more... | Lazy evaluation leads to unpredictable memory consumption and performance. |
| 7 | ...faster than ... | ...equally as fast... | ...faster than... | ...more... | ...less... | Hard to answer in the context of my experience where no immutable data structures are directly available |
| 8 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...less... | ...more... | Generally hard to compare, persistent data structures have different benefits than mutable ones. Runtime performance IMO less important than programmer productivity and program correctness. |
| 9 | ...slower than ... | ...faster... | ...equally as fast as... | ...more... | ...an equal amount of... | |
| 10 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | |
| 11 | ...equally as fast as... | ...equally as fast... | ...faster than... | ...an equal amount of... | ...an equal amount of... | These questions are stupid to answer without context or more infos |
| 12 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...more... | ...more... | My code is mostly io bound so data structures are usually not critical |
| 13 | ...faster than ... | ...faster... | ...faster than... | ...less... | ...less... | |
| 14 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | The correct answer to all this questions is: It depends. |
| 15 | ...equally as fast as... | ...faster... | ...equally as fast as... | ...more... | ...more... | Seem very generalizing; the questions all fit nearly all answers depending on context. |
| 16 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...more... | ...more... | it depends |
| 17 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...more... | ...more... | It highly depends on the context, in reality |
| 18 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...more... | ...an equal amount of... | |
| 19 | ...slower than ... | ...faster... | ...slower than... | ...an equal amount of... | ...an equal amount of... | Memory allocation depends on garbage collection–good GC will keep memory usage roughly equal, with the exception of during some recursive methods. The question is somewhat too broad though, and depends a lot on whether or not the programmer knows how to or wishes to optimize for these parameters. |
| 20 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | Yes: garbage collection versus accurate memory management. For system programming, I still want C++. I don't want a garbage collector whenever energy consumption matters. Functional language must rely on garbage collection, which is a no-go in these kinds of situations (e.g. embedded). |

Table B.4.: Performance part 1

| Id | Programs written in functional programming languages tend to run _____ corresponding imperative implementations. | Functional programming languages using lazy over strict evaluation tend to run _____ . | Immutable data structure operations tend to run _____ corresponding imperative implementations. | Programs written in functional programming languages tend to allocate _____ memory than/as corresponding imperative implementations. | Immutable data structures use up _____ memory than/as corresponding imperative implementations. | Do you have comments on the topic performance that have not been covered in this section? |
|---|---|---|---|---|---|---|
| 21 | ...slower than ... | ...equally as fast... | ...faster than... | ...more... | ...an equal amount of... | |
| 22 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...an equal amount of... | ...more... | |
| 23 | ...equally as fast as... | ...faster... | ...slower than... | ...more... | ...more... | |
| 24 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | The performance is influenced by many other factors. The most important factor is the developer's tendency to avoid thinking. Therefore, as far as it concerns me, the programming paradigms and the mutability of data structures do not influence the performance. (Humorous anecdote: Just compare typical C++ and Java applications when it comes to memory usage. When the kernel's OOM killer acts due to a lack of memory on a server of the company I work at, it is always a java process that dies.) |
| 25 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...more... | ...more... | So far I haven't had to concern myself with a deeper understanding of differences in performance between imperative and functional paradigms, at least nothing more from what can be "fixed" by using lazy linked lists. |
| 26 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...more... | ...more... | |
| 27 | ...equally as fast as... | ...faster... | ...faster than... | ...more... | ...an equal amount of... | |
| 28 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | I did not do any micro-benchmarks. Fundamentally changing the data structure also helps. |
| 29 | ...slower than ... | ...faster... | ...slower than... | ...more... | ...an equal amount of... | |
| 30 | ...equally as fast as... | ...slower... | ...equally as fast as... | ...more... | ...more... | |
| 31 | ...equally as fast as... | ...slower... | ...slower than... | ...more... | ...more... | |
| 32 | ...equally as fast as... | ...faster... | ...faster than... | ...more... | ...an equal amount of... | |
| 33 | ...slower than ... | ...faster... | ...faster than... | ...more... | ...more... | |
| 34 | ...equally as fast as... | ...faster... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | Python, Java and javascript are all inefficient in their own way, so haskell compares against them pretty well. How much time is spent on optimizing is more important than the language in my experience. |
| 35 | ...equally as fast as... | ...equally as fast... | ...slower than... | ...more... | ...more... | |
| 36 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | answer "equal" really means "don't know, don't care" |
| 37 | ...slower than ... | ...equally as fast... | ...faster than... | ...more... | ...more... | |
| 38 | ...slower than ... | ...equally as fast... | ...slower than... | ...more... | ...more... | |
| 39 | ...slower than ... | ...equally as fast... | ...slower than... | ...more... | ...more... | |
| 40 | ...equally as fast as... | ...slower... | ...slower than... | ...more... | ...more... | |

Table B.5.: Performance part 2

| Id | Programs written in functional programming languages tend to run _____ corresponding imperative implementations. | Functional programming languages using lazy over strict evaluation tend to run _____ . | Immutable data structure operations tend to run _____ corresponding imperative implementations. | Programs written in functional programming languages tend to allocate _____ memory than/as corresponding imperative implementations. | Immutable data structures use up _____ memory than/as corresponding imperative implementations. | Do you have comments on the topic performance that have not been covered in this section? |
|---|---|---|---|---|---|---|
| 41 | ...equally as fast as... | ...slower... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | Harder to predict space behaviour for lazy languages than for strict ones. Still not solved. |
| 42 | ...equally as fast as... | ...faster... | ...faster than... | ...more... | ...an equal amount of... | |
| 43 | ...equally as fast as... | ...equally as fast... | ...equally as fast as... | ...an equal amount of... | ...an equal amount of... | |

Table B.6.: Performance part 3

| Id | In theory, purely functional programs can be almost arbitrarily parallelised. In practice, it is therefore very easy to parallelise your programs written in a functional language. | If you selected other, please state your custom answer. | The speedup gained from parallelising functional programs is _____ compared to the speedup of parallelising imperative ones. | Do you have comments on the topic performance that have not been covered in this section? |
|---|---|---|---|---|
| 1 | TRUE | | ...bigger... | |
| 2 | FALSE | | ...smaller... | |
| 3 | Other... | theoretically | ...about the same... | |
| 4 | Other... | first part true, second part false | ...smaller... | |
| 5 | FALSE | | ...about the same... | |
| 6 | TRUE | | ...about the same... | |
| 7 | TRUE | | ...about the same... | Hard to answer, should provide a don't know |
| 8 | TRUE | | ...bigger... | |
| 9 | TRUE | | ...about the same... | |
| 10 | TRUE | | ...bigger... | |
| 11 | TRUE | | ...bigger... | |
| 12 | FALSE | | ...smaller... | Parallel speed up comes from data parallelism not from programming lang |
| 13 | TRUE | | ...about the same... | |
| 14 | TRUE | | ...about the same... | |
| 15 | Other... | Depends greatly on the program in question. Calculating fibonacci does not parallelize at all, doing a sort will parallelize very well. | ...bigger... | |
| 16 | TRUE | | ...about the same... | it depends |
| 17 | TRUE | | ...bigger... | |
| 18 | TRUE | | ...bigger... | |
| 19 | TRUE | | ...bigger... | |
| 20 | TRUE | | ...about the same... | Yes: big win for FP in this area IMHO. Extremely important for the future due to the multiplication of cores. The very reason why I study category theory! |
| 21 | TRUE | | ...about the same... | |

Table B.7.: Parallelism part 1

| Id | In theory, purely functional programs can be almost arbitrarily parallelised. In practice, it is therefore very easy to parallelise your programs written in a functional language. | If you selected other, please state your custom answer. | The speedup gained from parallelising functional programs is _____ compared to the speedup of parallelising imperative ones. | Do you have comments on the topic performance that have not been covered in this section? |
|---|---|---|---|---|
| 22 | TRUE | | ...about the same... | |
| 23 | TRUE | | ...about the same... | |
| 24 | FALSE | | ...about the same... | The difference here is most definitely not in running time, but in development time. |
| 25 | Other... | Haskell has very elegant constructs for these, the same are not entirely true for Scala, but so far it showed itself to be easier in general than it was in Java. | ...about the same... | I don't know the correct answer for the last question. My honest answer would be too long and too handwavy. |
| 26 | TRUE | | ...bigger... | |
| 27 | TRUE | | ...bigger... | |
| 28 | TRUE | | ...about the same... | |
| 29 | TRUE | | ...about the same... | |
| 30 | Other... | pure/impure | ...about the same... | |
| 31 | Other... | No experience | ...about the same... | |
| 32 | Other... | Not always, depends on the problem | ...about the same... | |
| 33 | TRUE | | ...about the same... | |
| 34 | TRUE | | ...about the same... | |
| 35 | TRUE | | ...about the same... | |
| 36 | TRUE | | ...bigger... | |
| 37 | TRUE | | ...bigger... | |
| 38 | Other... | Certainly some programs can be easily parallisable, but in the general case it is very hard to find an optimal split for the work. Haskell does provide good libraries for parallising pure computations and playing with alternative strategies or even running the same code on GPUs. This is something that would be very difficult to pull of with imperative languages | ...about the same... | the last one depends a lot on what one means with imperative programs: functional code will often compile to the same code if it is simple enough. Imperative code can be slower if there is shared memory usage requiring memory barriers etc. It can go either way depending heavily on cache usage. |
| 39 | TRUE | | ...bigger... | |
| 40 | TRUE | | ...bigger... | |
| 41 | FALSE | | ...about the same... | It's very application dependent |
| 42 | TRUE | | ...bigger... | |
| 43 | TRUE | | ...about the same... | |

Table B.8.: Parallelism part 2

| Id | Functional programming is lacking the critical mass to be seen as relevant. | If you selected other, please state your custom answer. | The tools/library/infrastructure/IDE support for functional programming languages is _____ . | There is enough education provided in functional programming. | If you selected other, please state your custom answer. | The community support for functional programming languages is _____ . | Do you have comments on the topic tooling that have not been covered in this section? |
|---|---|---|---|---|---|---|---|
| 1 | TRUE | | ...very bad... | FALSE | | ...ok... | |
| 2 | TRUE | | ...ok... | FALSE | | ...ok... | |
| 3 | FALSE | | ...not so good... | TRUE | | ...ok... | |
| 4 | TRUE | | ...very bad... | FALSE | | ...ok... | |
| 5 | TRUE | | ...ok... | TRUE | | ...excellent... | |
| 6 | TRUE | | ...not so good... | FALSE | | ...excellent... | |
| 7 | TRUE | | ...very bad... | FALSE | | ...not so good... | |
| 8 | FALSE | | ...good... | FALSE | I guess this depends, there is a lot of material available, but is it also used? | ...excellent... | |
| 9 | FALSE | | ...good... | TRUE | | ...good... | |
| 10 | FALSE | | ...excellent... | TRUE | | ...excellent... | |
| 11 | FALSE | | ...not so good... | FALSE | | ...good... | |
| 12 | FALSE | | ...ok... | TRUE | | ...not so good... | |
| 13 | TRUE | | ...not so good... | FALSE | | ...good... | |
| 14 | Other... | mass of what? I believe it simply lacks the people that have enough skill and experience in programming and designing functional applications. | ...good... | FALSE | | ...good... | An option, I don't know would be nice. I never worked with a pure functional programming language outside of class. So, I just don't know. |
| 15 | FALSE | | ...good... | Other... | There needs to be a much greater focus on what makes functional programming a *style* rather than a language characteristic. The quality needs to go up, not the quantity. | ...good... | |
| 16 | TRUE | | ...ok... | FALSE | | ...ok... | |
| 17 | FALSE | | ...ok... | FALSE | | ...good... | |
| 18 | TRUE | | ...very bad... | FALSE | | ...not so good... | |
| 19 | Other... | Varies from industry to industry, but it's seeing growth all over. React and similar projects are even bringing it to web development. In data pipeline engineering, it seems to be more the rule than the exception. | ...good... | FALSE | | ...good... | |

Table B.9.: Tooling & community part 1

| Id | Functional programming is lacking the critical mass to be seen as relevant. | If you selected other, please state your custom answer. | The tools/library/infrastructure/IDE support for functional programming languages is _____ . | There is enough education provided in functional programming. | If you selected other, please state your custom answer. | The community support for functional programming languages is _____ . | Do you have comments on the topic tooling that have not been covered in this section? |
|---|---|---|---|---|---|---|---|
| 20 | TRUE | | ...not so good... | FALSE | | ...ok... | FP should crush OOP in the coming decades. Exactly like when OOP took over imperative.One day soon, IMHO, FP will become trendy. And then, OOP will be old-fashioned.It will come from teachers in college (new gen of devs understanding FP) and current/future startups.It took 20 years for Design Patterns to be broadly known and applied. Some time is needed to master categories... my 2 cents! |
| 21 | TRUE | | ...ok... | TRUE | | ...ok... | |
| 22 | FALSE | | ...good... | FALSE | | ...good... | |
| 23 | FALSE | | ...good... | TRUE | | ...good... | |
| 24 | FALSE | | ...excellent... | FALSE | | ...good... | |
| 25 | Other... | Nah, what we lack is back education in fundamentals. As hand-wavy example: The courses that touch turing machines are too few and the ones that dare talking lambda calculus even less. | ...good... | FALSE | | ...ok... | |
| 26 | FALSE | | ...excellent... | FALSE | | ...excellent... | |
| 27 | FALSE | | ...excellent... | FALSE | | ...excellent... | |
| 28 | FALSE | | ...good... | TRUE | | ...good... | |
| 29 | FALSE | | ...good... | FALSE | | ...good... | |
| 30 | FALSE | | ...ok... | FALSE | | ...excellent... | |
| 31 | TRUE | | ...good... | FALSE | | ...good... | |
| 32 | FALSE | | ...not so good... | FALSE | | ...excellent... | |
| 33 | TRUE | | ...very bad... | FALSE | | ...excellent... | |
| 34 | Other... | It's starting to get there with functional feature creeping into Java (Streams, Optional) and Javascript | ...ok... | FALSE | | ...good... | |

Table B.10.: Tooling & community part 2

| Id | Functional programming is lacking the critical mass to be seen as relevant. | If you selected other, please state your custom answer. | The tools/library/infrastructure/IDE support for functional programming languages is _____ . | There is enough education provided in functional programming. | If you selected other, please state your custom answer. | The community support for functional programming languages is _____ . | Do you have comments on the topic tooling that have not been covered in this section? |
|---|---|---|---|---|---|---|---|
| 35 | TRUE | | ...not so good... | FALSE | | ...good... | |
| 36 | TRUE | | ...excellent... | Other... | No idea | ...ok... | Can't evaluate community support without rigorous study. |
| 37 | FALSE | | ...ok... | FALSE | | ...excellent... | |
| 38 | Other... | I think FP has gained the critical mass, but necessarily manifested via functional programming languages. The adoption of functional techniques and immutable data structures is definitely on the rise (react etc) | ...not so good... | Other... | There's a lot of good material out there, but perhaps not enough work done at teaching FP at universities | ...excellent... | |
| 39 | FALSE | | ...not so good... | FALSE | | ...good... | |
| 40 | TRUE | | ...very bad... | FALSE | | ...good... | |
| 41 | FALSE | | ...ok... | Other... | It's complicated: the difficult point is going from the principles to medium sized projects | ...good... | |
| 42 | TRUE | | ...not so good... | FALSE | | ...excellent... | |
| 43 | FALSE | | ...ok... | TRUE | | ...excellent... | |

Table B.11.: Tooling & community part 3

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 1 | Yes | I feel that if the required knowledge of functional programming was present, many problems in industry could be solve more elegantly and reliably (better correctness & maintainability) | Better training. Better tools. | Functional programming is often seen as something esoteric, and the user groups do not make this better. I feel that this can be made more accessible. | |
| 2 | Yes | | | | |
| 3 | No | It's not the way we think | use multi paradigm languages | | |

Table B.12.: General remarks part 1

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 4 | Yes | I have seen quite a lot of code written in imperative languages, that was almost purely functional. All of it could have been implemented in a functional language, reducing a lot of syntax overhead, making the intend of the code much clearer. | I feel that one of the most important parts is higher education. Only recently have I seen functional programming gaining traction outside of "purely academic" institutions like ETH. I strongly believe that students should be taught on functional programming, as a paradigm, even at the same time as they are learning OOP or at most one semester later. I think that this would help them getting a firm grasp on the concepts of FP, thus motivating them to use/try it during their school projects, thus familiarizing them even more with FP. | | Just remember, a monad is just a monoid in the category of endofunctors. |
| 5 | Yes | A little bit more courage to do so ... | More success stories. | No, I simply like it. | I think that functional programming is on a good track. But these things tend to need really a long long time. Im am enthusiastic about functional programming since about 25 years, and I always wondered why people voluntarily used object-oriented languages ... |
| 6 | Yes | Functional languages allow for more powerful abstractions and are therefore very suitable for high-level code. Especially languages which have a strict type system, like Haskell and Idris, make it easier to write correct code. | Teach the fundamental concepts of FP in schools, but don't stop there. Show how FP can be used to solve real world problems. | What's the best way of approaching software architecture and design, when working with FP? How does it differ from established approaches? | |
| 7 | Yes | | Monads to the masses | Are monads the only reasonable way to model necessary side effects? | |
| 8 | Yes | It's a very good fit for modern stateless request-processing type of work. | Teach it, I wouldn't be surprised if functional programming is much easier to grasp for a mathematically adept pupil than imperative programming. | Is a monad just like a burrito? | |
| 9 | Yes | | | | |
| 10 | Yes | | | | |

Table B.13.: General remarks part 2

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 11 | Yes | More training, time to evaluate, to much oop evangelists | trainings | WTF IS A MONAD, LENS, BANANA OR A FUNCTOR? | |
| 12 | Yes | Easier io is key. | Approachable solutions to day to day problems, that money can almost copy past from slashdot. | | |
| 13 | Yes | especially in combination with big data | teaching functional programming from begin. not teaching OO as the 'standard' | are there any software patterns in FP? | |
| 14 | Yes | It's just never considered as option. Mostly because companies tend to use the same language for all projects. But there is no language that is the best tool for all projects. In the end, you end using a good tool for most projects instead of the best for every project. I believe this is strictly coupled with the lack of experienced developers or the fact, that it is hard to find skilled people that have experience in more than 2 programming languages — not to mention programming paradigmas! and make it a mandatory subject in computer science studies. But please, don't use an exotic language that never has a chance in practice. Use something with a good community, broad availability of documentation, e.g. Haskell (uh.. thats controversial!) | What the heck are monads ;) | | |
| 15 | No | Functional programming as a paradigm needs more widespread knowledge and use. Functional languages will follow as an easier way to write the same software afterwards. | This is the wrong question to ask. You want to spread the paradigm of functional programming, not the language(s) themselves. | | Did you look at people using functional programming outside of the closed-space of functional-only programming languages? |
| 16 | Yes | Companies should not restrict themselfs to one single language. Instead, always evaluate for the one that fits most. | In my experience, the tools/languages used by the engineers in a company are selected by people which haven't developed a single software in the last years (10+). Thus, they have no idea about current trends and aren't open for new things (like: do not change a running system). These people have to be targetted as the actual developers have probably not much of a choice. | | |



Table B.14.: General remarks part 3

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 17 | Yes | Any code can benefit from having a declarative paradigm, given that code always has to be maintained | Be taught in school before imperative paradigms and mutability. Functional programming is closer to math and closer to how humans think, which should be more familiar as one's first programming language. | | |
| 18 | Yes | | | | |
| 19 | Yes | This is purely a matter of preference. I like to work in FP style, so I hope it becomes more pervasive. | The whole FP community could be less distant and condescending to OOP developers and newbies. The dogmatic way of discussing design style is extremely off-putting, even to someone like me who has been doing this for a long time. | | |
| 20 | Yes | Most people I know have never heard about FP. Or for them, it is just a buzz word.Some people even believe that JS is a FP language!Many of my current coworkers are interested in F#, because they are MS fanboys. But none of them understand categories or are willing to learn it... | Teach students. Make sure students understand the value of FP.Some of them will create startups. Some of these startups will be the new GAFAs. And that will start the trend.New tech does not come from established companies (I learnt that the hard way). Let them die! | Nope. I struggle with category theory and with Haskell syntax. The learning curve is steep. And this is the reason why students are a better target. Because seasoned programmers like me have forgotten about mathematics! | FP will eventually rule in most situations (but those where C/C++/OpenCL will still rule).OOP was a way to create business abstractions that simplify reasoning about a large code base. FP gives the ability to write correct programs thanks to the power of mathematics, and to parallelize programs thanks to purity. This is the way I see it today... |
| 21 | Yes | | | | |
| 22 | Yes | | | | |
| 23 | Yes | | | | |

Table B.15.: General remarks part 4

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 24 | Yes | I am heavily pushing code in functional style at work. As far as this is possible in javascript, anyway. The resulting code is always smaller, better abstracted, better reusable, easier to understand, and easier to test. Mostly, the functional code is the result of untangling a stateful, imperative mess. | Selective breeding of developers, I would say. Effective use of functional programming requires an excellent ability to abstract. (Did someone mention monads?) It is really difficult. Otherwise, please don't spread it. As long as idiots stay away from it (unlike it happened to Perl, JavaScript, and Java), functional programming will remain a developer's heaven. | I often ask myself how far one can replace testing by proving the correctness of an application. Functional programming languages tend to have excellent type systems, which already go a long way. Some even have built-in theorem provers that can verify even more properties, sometimes with a little help of the developer. I wonder how effective this would be on a daily basis for the average application and developer. (It certainly would be even more demanding of the developer than functional programming itself.) | I am of the opinion that overall, functional programming always wins against imperative or object oriented programming in terms of maintainability, code reuse, and development time. I am not yet quite clear about this, but I think that functional languages support abstraction (and therefore code reuse) far better than, say, OOP. The reason might be that each OOP language has one fixed set of abstractions provided by unhealthy amounts of syntactical sugar. Functional programming languages, however, seem to allow the developer to do their own abstraction, without any syntactical sugar at all. Likely, the reason is that the abstraction required in software development is of mathematical nature, and that functional programming languages are more mathematical than any OOP language you might find. And, to make this point clear, abstraction is at the heart of software development. It is done on all levels of an applications, from the topmost view the user has, down to the single data structures and the operations on it. This is also the reason why software development is no engeneering discipline. If you encounter someone calling herself "software engeneer", run away as fast as you can. (Now, this was a little bit on the philosophical side, which is very important for me in my daily work. Coworkers may not agree, though. And on a side note: Don't forget about other paradigms like logic programming. They are also very important and deserve their place.) |

Table B.16.: General remarks part 5

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 25 | Yes | In general is a lot easier built up on good foundations. That's what FP provides. And I don't mean that on any particular language. You do FP even in very verbose Ruby or JS. If the language gives some syntax that helps with that, great! | | Oh yeah, way too many to ask here, way too advanced to ask here :). One that is not that advanced: encodings of continuations in lambda calculus and in some abstract machines. | Yeah, "Do you have comments on the topic performance that have not been covered in this section?" not sure what "topic performance" is supposed to mean. I took it to mean "this topic", which seemed to make more sense. |
| 26 | Yes | They should be and are starting to be. | A complete redo of the undergraduate computer science curriculum in most universities. | How can we better deal with targeting embedded systems like AVR in functional programming? | |
| 27 | Yes | | | | |
| 28 | Yes | FP is still new-ish, people lack experience, it is a risk for new projects. There needs to be at least one expert who helps introducing and teaching about the concepts. | scala-exercises and the Typelevel community are good examples, structured and open communities with a well designed community process. | It still seems quite new, there are more than one way to do something. Sometimes the purest way is also quite unecessarily complex. Basic FP paradigms as map/flatMap/fold, pattern matching, types are fundamental building blocks. For higher level architectural things some people say, use free monad, others say use monad transformers/MTL. Then impure things such as local mutability are sometimes useful, vs purely functional. | Good survey, keep it up |
| 29 | Yes | I too few programmers know about functional programming or have experience with it. | Teaching professional programmers functional programming and the benefits of a functional approach in sofware engineering should help spread its use. | No | |
| 30 | Yes | | | | |
| 31 | Yes | | Change the Haskell motto! | | |
| 32 | Yes | In my opinion they allow faster development cycles and more work done per person | Solving the problems we have in our areas and "evangelizing" people from those areas that use other languages. For example, I organized datahaskell.org and started spreading it not in Haskell meetups, but rather data science ones. | | |
| 33 | Yes | | IDE support, more oibraries, but they will come, advanced courses | | |
| 34 | Yes | More projects just need to bite the bullet and prove that programmer productivity is greater than with traditional approaches. | More case studies / good testimonials. | | |

Table B.17.: General remarks part 6

| Id | Do you think that functional programming languages could be more often used in practice (industry/enterprise projects)? | Please elaborate on your previous answer. | What could/should be done to spread the use of functional programming languages? | Is there an aspect or open question about functional programming that has been bothering you and you would like to be answered? | Do you have comments on this survey that have not been covered? |
|---|---|---|---|---|---|
| 35 | Yes | | | | |
| 36 | Yes | Yes, sir. | Nothing | | |
| 37 | Yes | | | | |
| 38 | Yes | FP programs are often developed quicker, have less bugs and are much more maintainable. It definitely makes sense to see wider industry adoption | Improve tooling and documentation | | |
| 39 | Yes | It is more maintainable | Alter the perception of its usability: easy getting started (e. g. online IDE), teaching it with practical examples, success stories like facebooks haskell group) | | |
| 40 | Yes | | | | |
| 41 | Yes | Why not? | Keep going: we're getting there! | | |
| 42 | Yes | | More tools (in particular better IDEs), more good/well-supported libraries, easier entry for newcomers | Haskell in particular is not marketing itself well. A top google hit has the title "Haskell is useless". Useless indeed for spreading the word. | |
| 43 | Yes | I have worked on 3 very successful Haskell projects in industry over the last 15 years. I doubt they could have been done in any other language. | Find sponsors to fund improving the infrastructure | Haskell should be a really good language for numerical computation but sadly only a few people work on this area. This bothers me. I'd like to see more people contributing to this area. | |

Table B.18.: General remarks part 7

# C. Interview with Cyrill Schenkel

Cyrill Schenkel is a fellow student of the author who, at the same time the thesis took place, wrote his bachelor thesis on modern IDE support for functional programming [48]. To identify the requirements for his IDE, he conducted several interviews and conducted research on library, IDE and community support for the language Haskell. The author conducted an interview with Cyrill Schenkel in order to share results. A short transcription of the interview (translated into English) is given in this chapter:

MARIO MEILI: Does Haskell offer the same amount of standard libraries than for example Java, or is it significantly less?

CYRILL SCHENKEL: The amount of libraries is about the same. The problem lies in the quality of the libraries. For example, there is no platform independent UI library that is usable. Also, it is not advised to use the Haskell standard strings. This is well known in the community. One can find almost no logging library using standard strings. In general, it can be said that the Haskell standard library has historically grown and was never broken for to preserve backwards compatibility. Also, naming conventions for libraries were introduced in a later point in time, so there are old libraries that do not conform. Because of that, almost all companies define their own prelude. Libraries that are important are MTL, containers and STM. Another problem is the unsupervised library repository. Everyone can upload their packages which results in a lot of unmaintained libraries. Sometimes people even upload personal backups. Libraries are therefore definitely a problem. However, there are attempts of the community to better this situation.

MARIO MEILI: How is the Haskell community support to better the issue?

CYRILL SCHENKEL: There is an IRC channel that is very active. Over 10'000 people are ready to answer questions. Usually, one can expect an answer to his question within five minutes. Not every community does as good. For simpler questions, there is even a beginners channel. In addition, there is a Reddit Haskell group that is very active. In the region, there is the Zürich user group. The Commercial Haskell group, which is a group composed of companies opened a GitHub organization, where modified preludes and other useful contributions can be found.

MARIO MEILI: How is the IDE support for Haskell?

CYRILL SCHENKEL: First of all there is Cabal, which works good but is a bit old. It has a good feature set (similar to autotools with dependency management). But it is actually more of a package manager than an IDE. As is Stack, a newer tool which makes snapshots for reproducible builds. Stack is one of the contributions of the Commercial Haskell group. Intero is an integration for Emacs, Visual Studio Code and more. It is actually a fork of the Glasgow Haskell compiler interpreter. It offers autocompletion and shows type errors, which Cabal and Stack do not. The Emacs environment is very well developed. It also offers formatting (using hstylish and hindent). The use of hlint even allows to do some small refactorings. The major disadvantages of Intero are that it needs Stack (old cabal projects might not work, one needs to set up a stack project for everything) and it is not good in handling special language extensions (such as template haskell). Sometimes, some correct code might throw an error. These were the most used "IDEs" for Haskell development. Other products are EclipseFP (discontinued), Leksah (very strange UI, not stable, written in Haskell), Haskell for Mac (stable and offers installer), GHCMod (fork of ghci), GHCIDE (fork of ghci, automatically reload changed modules), Hare (haskell refactoring tool for haskell 98) and HIE (Haskell IDE Engine). But all of these have major drawbacks when compared to modern IDEs for imperative languages.

MARIO MEILI: Are there refactoring or optimization tools? If no, are they needed?

CYRILL SCHENKEL: Refactorings are not needed very often. Mostly one needs simple refactorings like autocompletion, formatting, method extraction and renaming. There is an interesting paper on the topic called Refactoring Functional Programs [49].

# D. Agreement for Project Thesis

**Student**: Mario Meili
**Semester**: FS 2017
**Advisor**: Prof. Dr. Farhad Mehta
**Project Partner**: Institute for Software (IFS) HSR
**Project Start Date**: 20.02.2017
**Project End Date**: 28.07.2017
**ECTS-Credits**: 12 ECTS Credits

### Project Title

If Functional Programming Is So Great, Why Isn't Everyone Using It?

### Goals and Project Description

Functional programming has been claimed to be the answer to many of the problems faced by software engineering today. Nevertheless, the large-scale adoption of functional programming in the mainstream of industry has been, to put it mildly, sluggish.

Aim:
The main aims of this project thesis are:

1. To systematically evaluate the extent to which functional programming is being used in various industrial, commercial and practical settings as well as the advantages and disadvantages of using functional programming in these settings.

2. To propose and investigate reasons why the use of functional programming has been limited, especially in areas where its advantages are found to be clear.

3. To propose, discuss and prioritize measures that could be taken to increase the benefit gained using functional programming in various industrial, commercial and practical settings.

Method:
The following tasks and methods could be used as an initial impulse to work towards the aims above:

1. Definition of scope: "functional programming" as a paradigm, programming style, a set of programming languages, or all these together? Which problem domains (industrial, commercial and practical settings) are to be considered? Examples include SCADA systems, accounting, the world-wide web, systems software, etc. Problem domains with the highest occurrence and potential for improvement, but the lowest penetration of functional programming, are to be given precedence.

2. Meta-study over existing publications relevant to the problem statement. This may include, but is not limited to academic proceedings, company press releases, internet sites, blog posts, etc.

3. The construction of prototypes to illustrate the advantages of functional programming in various problem domains. For instance, to illustrate what functional design pattern could be used in a typical SCADA system and their advantages over the state of the art.

4. Personal interviews with experts in the field.

### Deliverables

- A technical report in English, describing the work done as part of this project.
- A scientific article in English, fit for publication in a conference proceedings, summarising the results of this project.
- A critical personal reflection on the project and a statement of originality.
- A DVD containing all artefacts produced as part of this project.
- A final oral presentation of results with discussion.

### Competencies to Be Gained (Professional, Methodological and Self-Competence)

- The ability to understand, reflect on, and present scientific results.
- The ability to postulate, develop and evaluate hypotheses systematically, using the scientific method.
- Contribute to the state of the art in the application of programming language theory and technology in the industry.

### Assessment Criteria

Per the module description SWSY_PJ:

1. Overall assessment
   Criteria: Originality, innovativeness and applicability of the project results. Achievement of all project goals.

2. Organisation and Execution
   Criteria: Formulation of the task description, project planning, planned and systematic execution of the project, independent thought, dedication and collaboration skills.

3. Report
   Criteria: Content, structure, presentation and language.

4. Presentation
   Criteria: Consideration of the target audience, language and content.

5. Content
   Criteria: Preliminary study, requirement analysis, design, complexity, and scope. Quality of the artefacts produced.