Bachelor Thesis Project

Evaluating cyclomatic complexity on functional JavaScript

Author: Jesper Håkansson, Sherief Badran
Supervisor: Tobias Ohlsson
Semester: VT 2016
Subject: Computer Science

Abstract

Bugs in software is a very common problem, code reviews can help to catch bugs early on and detect which code is the most complex and may introduce bugs but when the code base is very large it can be costly to review all the code. Cyclomatic complexity can be used to give an indication of how complex the system source code is and help the developers to select which code they should review. But when measuring cyclomatic complexity on code written according to the functional paradigm, McCabe's formula will not be sufficient since it is a formula most suitable for imperative code. Therefore we are making adaptations to a formula suited for pure functional languages in order to fit functional JavaScript. We are using an inductive empirical quantitative measurement method to calculate cyclomatic complexity on a directed graph implementation in order to define adaptations for functional JavaScript. Our results show a working adapted version of the formula. We have measured on a graph implemented in Haskell and on a corresponding functional JavaScript version which results in a cyclomatic complexity difference at only 0.375.

Keywords: Software metrics, McCabe's cyclomatic complexity, functional programming, imperative programming, JavaScript, Haskell, functional JavaScript, measuring cyclomatic complexity on JavaScript, measuring cyclomatic complexity on the functional paradigm.

Contents

1 Introduction

In the last years, JavaScript has evolved as a language with various higher order functions such as forEach, map, filter and reduce [1]. The JavaScript community have lately paid a lot of attention to techniques of programming according to the functional programming paradigm through many different libraries such as Reactive Extensions [2], Folktale [3], Ramda [4], Redux [5] and Cycle.js [6] which is a library that is built upon Reactive Extensions. All the libraries have a lot of stars, forks and contributors which shows that there is a large interest from the JavaScript community. Stars represent likes, forks are when people have forked a library to their account in order to make changes. Contributors are the number of people who have been working and committed code to the project. There is also a specification called Fantasyland for interoperability of common algebraic structures in JavaScript such as setoid, semigroup, monoids, functors, apply, applicatives, foldable, traversable, chain, monad, extend and comonad [7].

Since bugs in software is a very common problem, code reviews can help to catch bugs early on and detect which code is the most complex and may introduce bugs but when the code base is very large it can be costly to review all the code, all the time. Cyclomatic complexity can be used to give an indication of how complex the system source code is and help the developers to select which code they should review [8, 11]. We suspect there is a problem when applying McCabe's formula on functional JavaScript and therefore this thesis will investigate if McCabe's formula is sufficient for measuring on functional JavaScript and if not, come up with a new formula for functional JavaScript.

1.1 Previous research

Previous research regarding metrics on functional JavaScript is minor and we could not find research about functional JavaScript and cyclomatic complexity, but there is published research regarding software metrics and comparison with imperative languages and functional languages. Ryder [8] states even the following: "In general there there has been little activity in the field of software engineering for functional programming". But there is a small amount of work present.

Van den Berg [9] uses software metrics such as Halstead volume and cyclomatic complexity to measure the complexity of programs written by students in two different programming languages. The code is written in one classical imperative language, Pascal and in a functional language, Miranda.

In addition to metrics Van den Berg also use a panel of experts to evaluate the code that the students have written. His study shows that there was a correlation between the experts opinions and the metrics regarding readability for the imperative programs, and that the correlation was significantly lower for the functional programs. The experts did not agree on the readability of the functional programs and Van den Berg think this causes the low correlation.

Ryder [8] states that cyclomatic complexity is not only a measurement of the complexity of a program but it is also useful as a measure of testability. Ryder also claims that software testing is one of the hardest things in software engineering and that software measurement can help the developers writing code that is easier to test. Software measurement cannot explain to the developer how to change the program to improve testability, but it is still of help because it can help to point the developer in the right direction, towards where a program might be hard to test and where it the most needs to be overseen. For instance McCabe's cyclomatic complexity can measure the number of test cases needed for structured testing. It is only an indication of testability though. Cyclomatic complexity metric can also be used to assess the readability of a program.

1.2 Problem formulation

When McCabe's cyclomatic complexity is applied on code written in a multi-paradigm language like JavaScript accuracy is assumed to be decreased. The reason is that McCabe's complexity is a software metric most suitable for imperative programming [9]. With that in mind, this thesis will investigate how cyclomatic complexity can be adapted in order to produce more accurate results for functional code written in a multi-paradigm language like JavaScript.

1.3 Motivation

Software bugs are a problem in every software and to help catch these bugs early in the development process peer reviews or code inspections are used. In large systems time and money are limited and therefore reviews cannot be made on the whole code base, hence only some parts of the code is selected for review. Cyclomatic complexity can be used to give an indication of how complex the system source code is and help the developers to select which code they should review [8, 11]. Furthermore the identified complex parts of the code also indicate difficulties in code testability, this information comes in handy when evaluating and estimating required resources for testing a code base.

1.4 Research Question

RQ1	Is McCabe's cyclomatic complexity sufficient for measurement in JavaScript regardless of programming paradigm?
RQ2	How can cyclomatic complexity be calculated to get a similar value for code in functional JavaScript compared to corresponding code in a pure functional language?

In RQ1, if the gap in cyclomatic complexity is consistent between each measurement, then McCabe's cyclomatic complexity is insufficient.

This thesis have two propositions, McCabe's cyclomatic complexity measurement is insufficient for measuring on functional JavaScript and Van den Berg's cyclomatic complexity formula for Miranda can be applied on functional JavaScript without any major modifications.

1.5 Scope/Limitation

This thesis will focus on smaller chunks of code and will not evaluate cyclomatic complexity on large real world projects simply because the time frame for this thesis is limited.

At first we had plans to evaluate maintainability index on the functional paradigm but as mentioned earlier time is limited and therefore we did focus on one metric instead of all the metrics included in maintainability index.

1.6 Target group

The target group of this thesis is the research community within the domain of software metrics and how cyclomatic complexity applies in functional languages or multi paradigm languages such as JavaScript. Furthermore this thesis might be of use for developers who are interested in having a sufficient way of measuring complexity of their functional code.

1.7 Outline

Chapter 1 introduces the reader to the subject of this thesis, the problem, motivation, target group, the scope of this thesis and the research questions. Chapter 2 is a theory chapter which goes deep into the subjects of cyclomatic complexity with the different formulas, the imperative and the functional

paradigm. Chapter 3 presents the method, preparation and execution for both our research questions. Chapter 4 describes short about our implementations. Chapter 5 displays the results conducted from the method. Chapter 6 goes through our discussion and analysis. Chapter 7 presents our conclusions and future research.

2 Theory

2.1 Background

Many programming languages are moving towards functional features. Languages such as C#, Java and Python have in the latest versions been adapting different functional characteristics such as lambda functions [10] and higher order functions such as map, filter, reduce [11].

2.1.1 Functional Programming

It is not very clear what types of characteristics that defines a language to be functional but according to Ryder “functions are treated as first class values that can be passed around as function arguments, returned as results, and generally manipulated in the same way as any other built in values.” [8]. Pure functions are also characteristic for functional programming but it is not the case in all functional languages for example Scheme [12]. By pure functions we mean a function that does not perform any side-effects, such as having a global variable that changes in different functions or relies on execution order. Side-effects are also I/O operations and network requests. Though this does not mean that all functions in functional programming are side-effects free, instead in functional programming you have functions that either are pure or not and the developer must be clear about which functions that are impure. Functional languages generally are more honest with which functions that are impure unlike for instance imperative languages.

In functional programming, code is often written in a declarative style, which means code that it hides detailed operations. A classical example is the JavaScript higher order functions (e.g. forEach, filter, map, reduce) that define what they do by their name but hide how the work is done as opposed to the well known and more imperative for loop. Programming is done with expressions or declarations unlike imperative programming where instead programming is done using statements. Code 1.1 exemplifies how filter is declarative in contrast to the example in Code 1.2 where the filtering is not as obvious.

```
const split = (isEven, list) => [
  list.filter(n => isEven(n)), list.filter(n => !isEven(n))
]
```

```
const isEven = x => x % 2 === 0
```

Code 1.1 Functional JavaScript split function

```
const split = (xs) => {
  const obj = {even: [], odd: []}

  for (let i = 0; i < xs.length; i++) {
    if (xs[i] % 2 === 0) {
      obj.even.push(xs[i])
    } else {
      obj.odd.push(xs[i])
    }
  }
  return obj
}
```

Code 1.2 Imperative JavaScript split function

2.1.2 Imperative Programming

Alex Gyor et al. states the following about imperative code: "code written in an imperative style would inevitably have side effects incompatible with the functional style" [13]. This is clearly a different approach than the functional paradigm. The imperative programming paradigm also implies code consisting of statements where the code in detail describes how the work is done. Imperative statements like for instance the for-loop mutates state by incrementing a loop counter. Imperative programming is a style performed in object oriented programming where a core concept is the class in which effects are performed on a shared global state within the class. The imperative programming style somewhat stands in contrast to the declarative style described in 2.1.1.

2.1.3 Functional and imperative JavaScript

JavaScript is influenced by both imperative and functional languages such as C and Scheme [14]. Because of this it has been given both imperative as well as functional traits from both paradigms. The language has all the characteristics for an imperative language but it also has the characteristics for a functional language. All the characteristics stated in 2.1.1 and 2.1.2 are

present in the language, such as passing functions around, being treated as first class values and being able to make pure functions. Therefore JavaScript can be written in either paradigm.

2.1.4 Cyclomatic Complexity

Cyclomatic complexity is a software metric by which to measure the number of linearly independent paths through the code. More precisely McCabe's cyclomatic complexity [15] can be calculated according to the following definition:

$$MCC = |E| - |V| + 2p \textbf{ or } MCC = |E| - |V| + 1p$$

where $|E|$ is the number of edges, $|V|$is the number of vertices and p is the number of connected components. A connected component corresponds to a function or module in a program. A vertex is characterized by a statement (e.g. if, for, switch and case), an edge connect the vertices within the same module hence p represents for instance a module subroutine.

The definition consisting of $1p$ rather than $2p$ is used when a graph is strongly connected. In this case the graph consists of one more edge which is compensated with the constant value of 1 instead of 2.

The principle mentioned above can be visualized with a control flow graph (CFG) as in Figure 1.1.

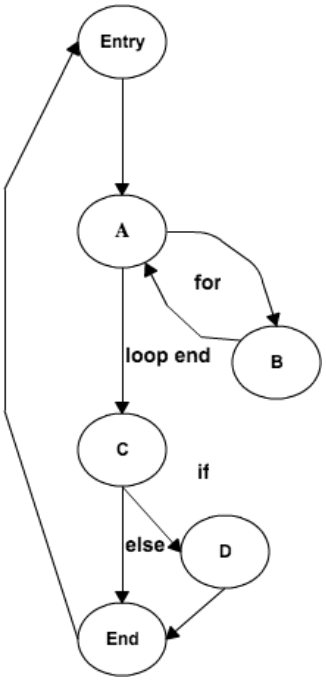

Figure 1.1 Control flow graph

Since the end vertex is connected back to the entry, MCC for the connected component and CFG in figure 1.1 is calculated as follows:

$$MCC = |E| - |V| + 1p, where\ p = 1$$
$$MCC = 8 - 6 + 1$$
$$MCC = 3$$

Alternatively when p = 1, the MCC can be calculated with a simplified formula derived from $MCC = |E| - |V| + 1p$.

Mills [16] proved that $E = 1 + \theta + 3\pi$ (where $\theta$is the number of functions and π is the number of predicates) and $V = \theta + 2\pi + 2$.

Substituting the expressions for E and V into $MCC = |E| - |V| + 2p$yields:

$$MCC = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2 = \pi + 1$$
$$MCC = \pi + 1$$

2.1.5 Cyclomatic Complexity for the functional paradigm

Van den Berg [9] has developed a way of measuring cyclomatic complexity based on McCabe's work but for the functional language Miranda. Based on control-flow models, Van den Berg analyzes the operational semantics for functions in Miranda in order to produce an adapted way of measuring cyclomatic complexity according to McCabe's principles. In order to answer RQ2, a way of measuring cyclomatic complexity on pure functional code has to be performed, translated and adapted for corresponding measurements in functional JavaScript. For this purpose Van den Berg's reasoning about adaptation of cyclomatic complexity to pure functional languages will be conducted. This thesis will use Haskell instead of Miranda which is no problem since Haskell is strongly influenced by Miranda. Van den Berg's principle of measuring cyclomatic complexity is selected since it is based on McCabe's definition, more precisely by determining the number of paths through code which Van den Berg also visualizes with control flow models. Below is a function and its corresponding control flow graph presented.

The split function takes two arguments, a function *p* which task is to return a boolean, true or false if the number is even or not. The second argument is a list of numbers. The split function will return a tuple with two lists, one list with all the even numbers and another list with all the odd.

```
split p [] = ([], [])
split p (x:xs) = if p x then (x : ys, zs)
  else (ys, x : zs)
  where (ys, zs) = split p xs


p :: Int -> Bool
p number = number `mod` 2 == 0
```

Code 1.3 Haskell split function

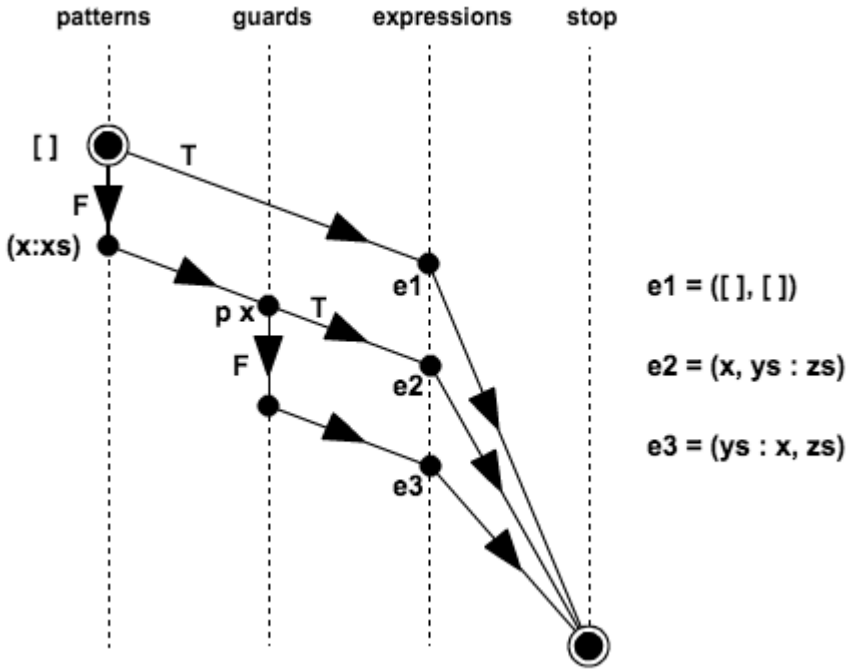

Figure 1:2 Control Flow Model

Van den Berg's complexity measure is defined as:

The total cyclomatic complexity number for a function is equal to 1 plus the sum of the left hand side, called LHS, plus the sum of right hand side, called RHS. The LHS consists of the sum of the pattern complexity. The pattern complexity is equal to the number of identifiers in the pattern, minus the number of unique identifiers in the pattern, plus the number of arguments that are not identifiers. The RHS consists of the number of guards, the number of logical operators, the number of filters in a list comprehension and the pattern complexity in a list comprehension.

This can be explained in the following equations:

Pattern complexity, LHS

$$LHS = Pattern\ complexity = pattern\ identifiers - unique\ pattern\ identifiers + non\ identifiers$$

RHS

$$RHS = Number\ of\ guards + Logical\ operators + number\ of\ filters\ in\ list\ comprehension + pattern\ complexity\ in\ list\ comprehension$$

Cyclomatic complexity

$Cyclomatic\ complexity\ =\ 1\ +\ LHS\ +\ RHS$

To illustrate the cyclomatic complexity measurement according to Van den Berg's definition we calculate the complexity on the split function defined earlier. Structuring the code to their corresponding sides.

LHS

```
split p []
split p (x:xs)
```

RHS

```
([], [])
| p x = (x : ys, zs)
| otherwise = (ys, x : zs)
where (ys, zs) = split p xs
```

In the LHS there are three pattern identifiers which are *p*, *x* and *xs*, there are two unique pattern identifiers which are *x* and *xs*, and there are two non identifiers which are *[]* and *(:)*.

On the RHS | *p x* is a guard. The number of logical operators, the number of filters in a list comprehension and pattern complexity are all 0.

$LHS\ =\ 3\ -\ 2\ +\ 2\ =\ 3$

$RHS\ =1$

$Cyclomatic\ complexity\ =\ 1+3+1=5$

The | *otherwise* guard does not count up the cyclomatic complexity since the *otherwise* keyword corresponds to an *else* statement which does not add an extra path through the code.

3 Method

Two graph implementations was used as a basis to conduct measurements on cyclomatic complexity in order to answer the research questions RQ1 and RQ2. These graphs has algorithms like depth-first search and transitive closure implemented. One graph was implemented according to the functional paradigm and the other one according to the imperative.

To answer RQ2 a directed graph was translated from parts of Haskell's Data.Graph core package into functional JavaScript. An implementation of a directed graph was chosen because it is a common concept in computer science and the different algorithms are commonly recognized.

First preparations, some information regarding implementations and execution of the method followed by execution examples to answer RQ 1 are described. The second part of the method focus on RQ2 and consists of preparations and execution followed by detailed examples of how measurements were conducted. Finally validity is discussed.

3.1 Scientific Approach

This method is performed by measuring cyclomatic complexity on code according to McCabe's theory and the cyclomatic complexity formula described in 2.1.5, resulting in an inductive empirical quantitative measurement method.

3.2 Method Description

The part of the method execution related to RQ1 will be referred to as *method execution 1 (ME1)* and likewise the method execution related to RQ2 will be referred to as *method execution 2 (ME2).*

3.2.1 Preparation for ME1

Jshint[1] has an implemented tool for measuring McCabe's cyclomatic complexity. The tool was forked and modified to not only calculate cyclomatic complexity for each function but also sum the total cyclomatic complexity for the whole program under measurement. In order to avoid Jshint only outputting warnings when cyclomatic complexity is to high, a flag was added to always obtain a resulting output. However, by cloning the repository containing our graph implementations the method can be carried out as described below.

[1] http://jshint.com/

In order to perform the RQ1 execution of the method, Node v5.9.1 and npm v3.7.3 are required to be installed.

3.2.2 Method Execution 1

In this execution cyclomatic complexity was measured on a graph program implemented in JavaScript according to the functional paradigm found in Appendix A1. A corresponding graph program was also implemented in JavaScript but according to the imperative paradigm found in Appendix A2.

The functional graph implementation will be referred to as FG and the imperative implementation of the graph will be referred to as IG.

- A measurement was conducted on FG and IG by manually calculating the cyclomatic complexity according to 2.1.4 on each function in the program. Each function's cyclomatic complexity was calculated with the formula $v = \pi + 1$, and then for each graph FG and IG the results were added together to obtain the cyclomatic complexity for each graph as a whole.

- The result from the step above was verified by recalculating the cyclomatic complexity for the whole program FG and IG by applying the function $v = E - V + 2p$.

- The Jshint tool was used to measure the total cyclomatic complexity of FG and IG in order to verify correctness of the results from the two steps above.

3.2.3 Measurement examples for ME1

The following examples demonstrate how measurements were manually performed on FG and IG respectively.

addNodes is a function from the IG program and is calculated as:

```
addNodes(pairs) {
  if (pairs instanceof Array) {
    for (let i = 0; i < pairs.length; i++) {
      this.addNodes(pairs[i])
    }
    return
  }

  this.edges.push(this.addNode(pairs))
}
```

Code 1.4 Imperative JavaScript addNodes function

According to McCabe's cyclomatic complexity defined in 2.1.4, both the if and for statements are paths through the code including the function main path which yields

$$v = \pi + 1 = 2 + 1 = 3$$

addEdges is a function from the FG program and is calculated as follows.

```
const addEdges = connect => Fnodes =>
    map(reduce(connect))(Fnodes)
```

Code 1.5 Functional JavaScript addEdges function

According to 2.1.5, the only path adding to the cyclomatic complexity is the function's main path.

$$v = \pi + 1 = 0 + 1 = 1$$

3.2.4 Preparation for ME2

By cloning the repository containing the graph implementation for Haskell found in Appendix A3 as well as the translated Haskell graph implementation into JavaScript found in Appendix A4 the method can be carried out as described below. The Haskell graph program is taken from parts of Haskell's Data.Graph package. The Haskell graph implementation will be referred to as HG and the corresponding JavaScript implementation will be referred to as TFG.

3.2.5 Method Execution 2

The steps below were iterated two times:

1. The concept and method for measuring cyclomatic complexity on the functional language Miranda was studied thoroughly. The theory is summarized in 2.1.5.

2. A graph was implemented in JavaScript, by translating the Haskell source code for the graph into functional JavaScript.

3. The way of measuring cyclomatic complexity in functional languages like Miranda and Haskell according to 2.1.4 was translated and adapted to the functional translated graph implemented in JavaScript according to 2.

4. A Measurement was conducted on the HG by manually calculating the cyclomatic complexity according the formula described in 2.1.4 on each function of the program.

5. A Measurement was conducted on TFG by manually calculating the cyclomatic complexity according to the formula described in 2.1.4 with adaptations for JavaScript described in 2.1.5.

6. The results from step 4 and step 5 was compared. The goal is for the results to converge. Evaluate and redo step 3 until the results from step 4 and 5 no longer converge.

7. Iterate any of the previous steps until the results of 6 evaluates to a satisfying result, which is that the results from 4 and 5 converged as near as possible.

- The first iteration was an attempt to execute step 3 above by defining how key properties like identifiers, unique identifiers, non-identifiers along with array comprehensions from the Haskell implementation are translated into JavaScript and counted.

- The second iteration was a re-execution of 5 but with the JavaScript helper functions excluded from the measurement of cyclomatic complexity. The helper functions are the functions that had to be implemented in JavaScript and correspond to functions built into Haskell.

3.2.4 Measurement examples for ME2

The following examples demonstrate how measurements were manually performed on HG and TFG respectively.

mapTable is a function from the HG program and is calculated as follows:

```
mapTable f t = array (bounds t) [ (,) v (f v (t!v))
| v <- indices t ]
```

Code 1.6 Haskell mapTable function

LHS:

Pattern identifiers:	**2 (f, t)**
Unique pattern identifiers:	**2 (f, t)**
Non-identifiers:	**0**

LHS = Pattern identifiers - Unique pattern identifiers + non-identifiers =
= 2 - 2 + 0 = 0

RHS:

Number of guards:	**0**
Logical operators:	**0**
Number of filters in a list comprehension:	**0**
Pattern complexity in a list comprehension:	**6**

Pattern complexity in a list comprehension is calculated with respect to the properties below:

Pattern identifiers:	**5 (v, f, t, !, indices)**
Unique pattern identifiers:	**2 (f, indices)**
Non-identifiers:	**3 (), (,), |)**

Note that "!" is an identifier since it is a Haskell infix notation function meaning that the function takes one argument on each side.

Pattern complexity in list comprehension = Pattern identifiers - Unique pattern identifiers + non-identifiers = 5 - 2 + 3 = 6

$Cyclomatic\ complexity\ =\ 1\ +\ LHS\ +\ RHS$= 1 + 0 + 6 = 7

mapTable is a function from the TFG program and is calculated as follows.

```
const mapTable = f => g => indices(g).map((v, i)
=> [v, f(g[i][v])])
```

Code 1.7 Functional JavaScript mapTable function

LHS:

Pattern identifiers:	**2 (f, g)**
Unique pattern identifiers:	**2 (f, g)**
Non-identifiers:	**0**

LHS = Pattern identifiers - Unique pattern identifiers + non-identifiers =
= 2 - 2 + 0 = 0

RHS:

Number of guards:	**0**
Logical operators:	**0**
Number of filters in a list comprehension:	**0**
Pattern complexity in a list comprehension:	**6**

Pattern complexity in a list comprehension is calculated with respect to the properties below:

Pattern identifiers:	**5 (g, v, i, indices, f)**
Unique pattern identifiers:	**2 (f, indices)**
Non-identifiers:	**1 ([])**

Pattern complexity in list comprehension = Pattern identifiers - Unique pattern identifiers + non-identifiers = 5 - 2 + 1 = 4

$Cyclomatic\ complexity\ =\ 1\ +\ LHS\ +\ RHS$= 1 + 0 + 5 = 5

3.3 Reliability and Validity

3.3.1 Internal validity

The implementation of FG and IG respectively including their associated algorithms has to correspond to each other in terms of returning equal output for equal input. In other words the imperative implementation of the depth first search has to return output equal to the functional implementation of depth first search provided that both are applied on the same graph.

Since the cyclomatic complexity of FG and IG were compared to each other, it is important that the implementation of the set of algorithms for each paradigm is conducted with equal time complexity T(n). The algorithms are implemented with an equally performed effort as possible to obtain quality and

best practices for each paradigm. The quality attributes in this context are to implement algorithms to have the minimum time complexity.
Furthermore since we lack experience and routine in programming with Haskell, an implementation by ourselves would have been a validity issue and therefore we instead chose to use parts of Haskell's Data.Graph core package.

In this thesis the graph translated from Haskell to a functional JavaScript version needs a corresponding imperative JavaScript implementation for the sake of a fair and optimized comparison of cyclomatic complexity between the two. Currently a comparison was made between the translated JavaScript graph and the imperative graph implemented for RQ1 which is not optimal since they differ in terms of implementation and functions on the graph does not really correspond to each other. On the other hand they are similar in terms of being a graph implementation and number of functions.

Furthermore all cyclomatic complexity measurements are done by the authors of this thesis. Even though measurements are done repeatedly and in an iterative fashion it is possible to make repeated mistakes caused by lack of external influences. Therefore it would have been preferable to let measurements be conducted by people external to this project provided that these people are familiar with the methodologies of the measurements. The results from other people conducting measurements could then have been compared and evaluated against measurements performed by the authors of this thesis. This way of improving the validity of measurements is suggested for a replicated study.

3.3.2 External validity

In the case of this thesis issues of external validity means that we don't have access to larger code bases of functional JavaScript, at least not from the JavaScript open source community.

It would have been desirable to perform measurements on a codebase larger than the code we performed the measurements on. The reason we chose to write functional JavaScript by ourselves is the lack of functional open source JavaScript projects of larger size and dignity.

3.3.3 Reliability

When reproducing the method of this thesis a risk is that measurements according to Van den Berg's approach or our approach of measuring cyclomatic complexity of functional JavaScript might (despite of our measurement examples) be misinterpreted. This is due to our lack of

experience with pure functional programming and especially Miranda as a language. A consequence would be that results are not possible to reproduce.

4 Results

The results are displayed below in tables and charts. In this section we use the $\underline{v}$ symbol which is a mathematical symbol for average.

4.1 Results for RQ1

Table 4.1.1 displays the cyclomatic complexity for the imperative implementation of a directed graph in the graph.js program.

Function name	Cyclomatic complexity
constructor Graph	2
addNodes	3
addNode	2
addEdges	3
addEdge	2
getNode	2
outDegree	2
inDegree	2
nodeCount	2
edgeCount	2
constructor Node	1
Program cyclomatic complexity	23

Table 4.1.1: Cyclomatic complexity for the imperative graph.js program.

The cyclomatic complexity for the whole program is 23. The number of functions in the program is 11.

Hence $\underline{v} = \frac{23}{11} \approx 2.090$

Table 4.1.2 displays the cyclomatic complexity for the functional implementation of a directed graph in the graph.js program.

Function name	Cyclomatic complexity
compose	1
curry	1
Graph	1
Graph.of	1
Graph.prototype.map	1
Graph.prototype.join	1
Fmap	1
findExistingNode	2
createNode	1
addNewNode	2
addEdge	1
Freduce	1
addEdges	1
degreeMap	1
degree	1
outDegree	1
inDegree	1
nodeCount	1
edgeCount	1
deepConvertItemsToNodes	1
convertItemsToNodes	1
graph	2
Program cyclomatic complexity	25

Table 4.1.2: Cyclomatic complexity for the functional graph.js program.

The cyclomatic complexity for the whole program is 25. The number of functions in the program is 22.

Hence $\underline{v} = \frac{25}{22} \approx 1.136$

Table 4.1.3 displays the cyclomatic complexity for the imperative implementation of a depth-first search algorithm in the dfs.js program.

Function name	Cyclomatic complexity
dfs	1
clearNodes	2
clearNode	2
dfsWithRoot	3
isCyclic	6
Program cyclomatic complexity	14

Table 4.1.3: Cyclomatic complexity for the imperative dfs.js program.

The cyclomatic complexity for the whole program is 14. The number of functions in the program is 5.

Hence $\underline{v} = \frac{14}{6} \approx 2.33$

Table 4.1.4 displays the cyclomatic complexity for the functional implementation of a depth-first search algorithm in the dfs.js program.

Function name	Cyclomatic complexity
dfs	1
clearNodes	1
clearNode	2
mapNode	1
dfsWithRoot	1
filterCirular	1
isCyclic	1
Program cyclomatic complexity	8

Table 4.1.4: Cyclomatic complexity for the functional dfs.js program.

The cyclomatic complexity for the whole program is 8. The number of functions in the program is 7.

Hence $\underline{v} = \frac{8}{7} \approx 1.14$

Table 4.1.5 displays the cyclomatic complexity for the imperative implementation of a transitive closure algorithm in the transitive-closure.js program.

Function name	Cyclomatic complexity
computeClosure	2
Program cyclomatic complexity	2

Table 4.1.5: Cyclomatic complexity for the imperative transitive-closure.js program.

The cyclomatic complexity for the whole program is 2. The number of functions in the program is 1.

Hence $\underline{v} = \frac{2}{1} \approx 2$

Table 4.1.6 displays the cyclomatic complexity for the functional implementation of a transitive closure algorithm in the transitive-closure.js program.

Function name	Cyclomatic complexity
transitiveMap	1
computeClosure	1
Program cyclomatic complexity	2

Table 4.1.6: Cyclomatic complexity for the functional transitive-closure.js program.

The cyclomatic complexity for the whole program is 2. The number of functions in the program is 2.

Hence $\underline{v} = \frac{2}{2} \approx 1$

Table 4.1.7 displays the cyclomatic complexity for the imperative implementation of the split function in the split.js program.

Function name	Cyclomatic complexity
split	3
Program cyclomatic complexity	3

Table 4.1.7: Cyclomatic complexity for the imperative split.js program.

The cyclomatic complexity for the whole program is 3. The number of functions in the program is 1.

Hence $\underline{v} = \frac{3}{1} \approx 3$

Table 4.1.8 displays the cyclomatic complexity for the functional implementation of the split function in the split.js program.

Function name	Cyclomatic complexity
isEven	1
split	1
Program cyclomatic complexity	2

Table 4.1.8: Cyclomatic complexity for the functional split.js program.

The cyclomatic complexity for the whole program is 2. The number of functions in the program is 2.

Hence $\underline{v} = \frac{2}{2} \approx 1$

4.2 Results for RQ2

Table 4.2.1 displays the cyclomatic complexity for the Haskell implementation of a directed graph in the graph.hs program.

Function name	Cyclomatic complexity
buildG	1
transposeG	1
reverseE	6
outdegree	1
indegree	1
edges	6
vertices	1
mapTable	5
Program cyclomatic complexity	22

Table 4.2.1: Cyclomatic complexity for the functional graph.hs program.

The cyclomatic complexity for the whole program is 22. The number of functions in the program is 8.

Hence $\underline{v} = \frac{22}{8} \approx 2.750$

Table 4.2.2 displays the cyclomatic complexity for the translated implementation from Haskell to JavaScript of a directed graph in the graph.js program with helper functions included.

Function name	Cyclomatic complexity
buildG	1
transposeG	1
reverseE	4
outdegree	1
indegree	1
edges	9
vertices	1
mapTable	7
accumArray	7
indices	1
head	1
compose	1
Program cyclomatic complexity	35

Table 4.2.2: Cyclomatic complexity for the translated functional graph.js program with helper functions included.

The cyclomatic complexity for the whole program is 35. The number of functions in the program is 8.

Hence $\underline{v} = \frac{35}{8} \approx 4.375$

Table 4.2.3 displays the cyclomatic complexity for the translated implementation from Haskell to JavaScript of a directed graph in the graph.js program.

Function name	Cyclomatic complexity
buildG	1
transposeG	1
reverseE	4
outdegree	1
indegree	1
edges	9
vertices	1
mapTable	7
Program cyclomatic complexity	25

Table 4.2.3: Cyclomatic complexity for the translated functional graph.js program.

The cyclomatic complexity for the whole program is 25. The number of functions in the program is 8.

Hence $\underline{v} = \frac{25}{8} \approx 3.125$

4.3 Average per program

Figure 4.3.1 displays the of average the measured cyclomatic complexity number per program for both the imperative and the functional JavaScript implementations using McCabe's formula described in 2.1.4.

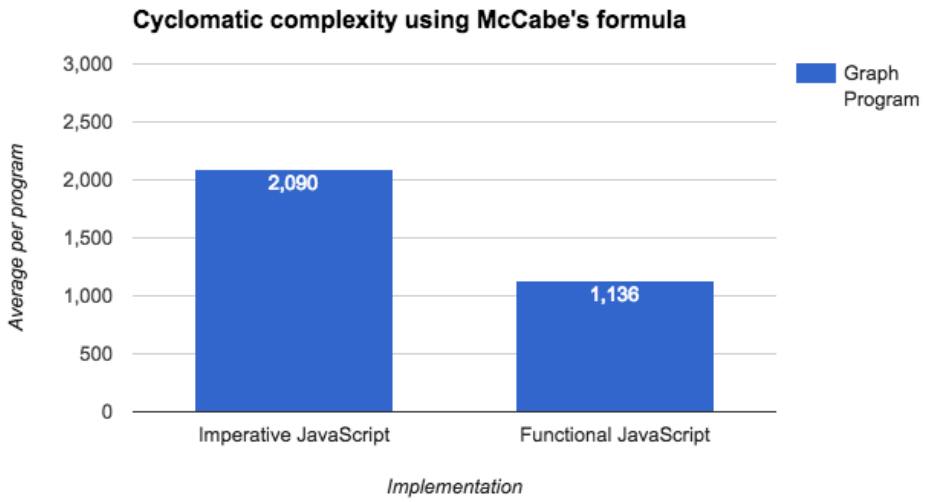

Figure 4.3.1 Average per program for both the imperative and the functional JavaScript implementations using McCabe's formula

Figure 4.3.2 displays the average of the measured cyclomatic complexity number per program for both the functional JavaScript and the Haskell implementations using the functional formulas described in 2.1.4 and 2.1.5 without the helper functions.

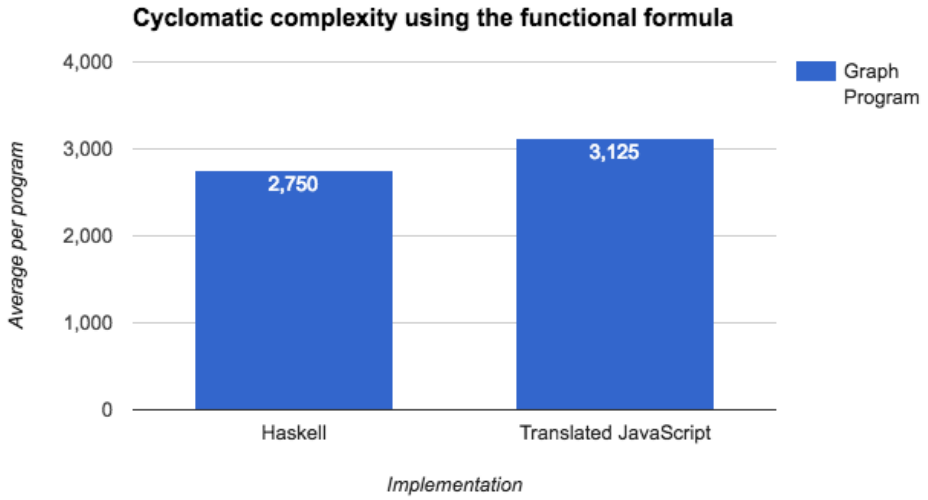

Figure 4.3.2 Average per program for both the functional JavaScript and the Haskell implementations using the functional formulas

Figure 4.3.3 displays the average of the measured cyclomatic complexity number per program for both the imperative and the translated JavaScript implementations but without the helper functions. The imperative implementation is measured with McCabe's formula described in 2.1.4 and the translated implementation is measured with the formula described in 2.1.5.

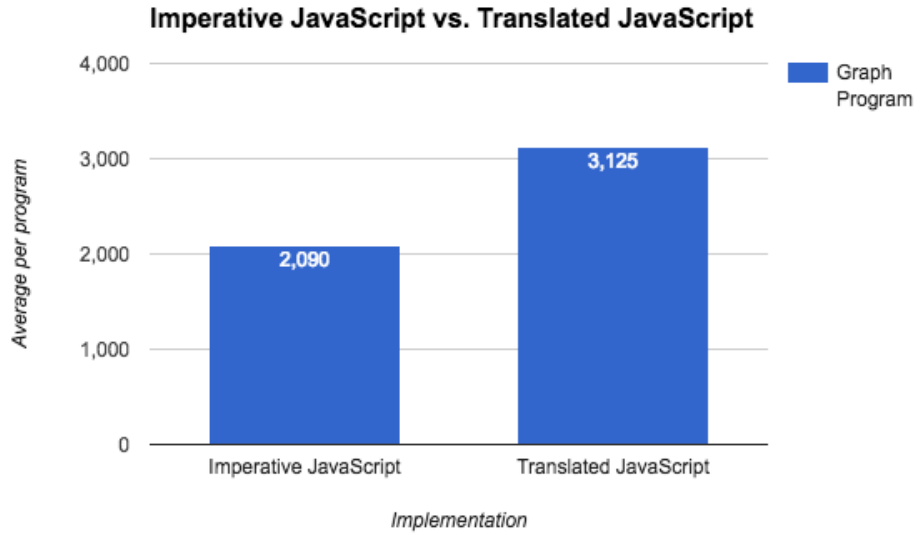

Figure 4.3.3 Average per program for both the imperative and the translated JavaScript implementations but without the helper functions

Figure 4.3.4 displays the average of the measured cyclomatic complexity number per program for both the functional JavaScript and the Haskell implementations using the functional formulas described in 2.1.4 and 2.1.5 with the helper functions included.

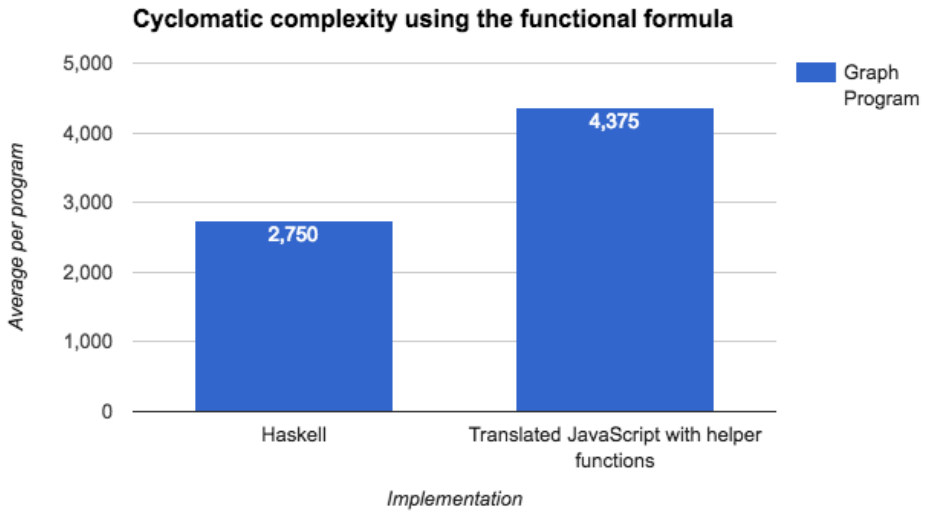

Figure 4.3.4 Average per program for both the imperative and the translated JavaScript implementations with the helper functions included

4.4 Cyclomatic Complexity for functional Javascript

The formula for measuring cyclomatic complexity in functional languages defined in 2.1.4 has been used and modified to fit JavaScript. Haskell has various functionality built into the language that JavaScript lacks such as pattern matching and list comprehensions and other various helper functions commonly used in functional programming.

In the LHS pattern identifiers are arguments in the function signature. The same applies for unique pattern identifiers. Non identifiers are array literals "*[]*", object literals "*{}*", the spread operator "*...*" as well as destructuring assignment "*{x, y} = {x: 1, y: 2}*". Non-identifiers of the same type that occur more than once are grouped into and counts up the cyclomatic complexity by one.

In the RHS guards correspond to if-statements, conditional (ternary) operator, and switch-statements. Logical operators are "*&&*", "*!*" and "||". The higher order function *.map()* on Array.prototype is treated as a list comprehension and *.filter()* as a filter in list comprehensions. For example in a *.reduce()* function we are outside the scope of list comprehensions and therefore is *.reduce()* treated as a function, which is an identifier.

5 Analysis and Discussion

Table 5.1.1 displays a summary of all the implementations average in cyclomatic complexity per function.

Implementation	$\underline{v}$
Imperative graph (IG)	2.090
Functional graph (FG)	1.136
Haskell graph (HG)	2.750
Translated JavaScript graph with helper functions	4.375
Translated JavaScript graph without helper functions (TFG)	3.125

Table 5.1.1: Summary of all the implementations average in cyclomatic complexity per function.

In this discussion, four different implementations of a directed graph will be discussed and from now on be referred to as stated below:

- **IG:** The imperative implementation of a directed graph program for RQ1 (Imperative Graph).
- **FG:** The functional implementation of a directed graph program for RQ1 (Functional JavaScript Graph).
- **TFG:** The functional implementation of a directed graph program that is translated from parts of Haskell's Data.Graph package (Translated Functional JavaScript Graph).
- **HG:** The Haskell implementation of a directed graph which is part of Haskell's Data.Graph package.

5.1 Discussion of method execution 1

In order to verify RQ1, namely that the current execution of measuring McCabe's cyclomatic complexity is insufficient when applied on functional JavaScript, measurements were conducted on both an imperative and a

functional JavaScript implementation of a directed graph. Measurements on the imperative implementation resulted in a cyclomatic complexity of 2.090and the functional implementation resulted in a cyclomatic complexity of 1.136, hence almost all functions in the functional program had an individual cyclomatic complexity value of 1 which clearly support the purpose of RQ1. In a pre-study conducted at the very beginning of this thesis, we used tools like JsComplexity[2] to measure the cyclomatic complexity on functional as well as corresponding imperative functions of different sizes, to verify the consistency of a gap in cyclomatic complexity between functional and imperative code written in JavaScript. Very often functional code snippets tend to have cyclomatic complexity values 1 or 2 while corresponding imperative implementations have 3-5. We think that this difference between the two paradigms is a big deal and makes McCabe's formula insufficient for measuring on the functional paradigm which makes our study interesting.

5.2 Discussion of method execution 2

According to table 4.2.2 the directed graph program was translated from parts of Haskell's Data.Graph package to functional JavaScript.

According to table 4.2.2 the TFG has a cyclomatic complexity of 4.375 including helper functions. Also a measurement on the same program was conducted but with helper functions excluded which according to table 4.2.3 resulted in a cyclomatic complexity of 3.125. We think the measurement excluding helper functions arrives at a fairer result since the helper functions we implemented in JavaScript correspond to functions that in Haskell are already built in.

The HG has a cyclomatic complexity of 2.750 compared to the corresponding and above presented result of 3.125 for TFG. The difference between the two is 0.375 which is acceptable since our goal was to come as close as possible after we translated Van den Berg's approach for measuring cyclomatic complexity in functional languages to be applicable in JavaScript.

The small difference of 0.375 indicates that our work with translating Van den Berg's approach to JavaScript is sufficient to be applied with the purpose of comparing the cyclomatic complexity of functional JavaScript with imperative JavaScript.

Furthermore it is highly relevant to also compare the cyclomatic complexity of TFG with the IG. The first is measured with our transformation of Van den

[2] http://jscomplexity.org

Berg's approach (for measuring cyclomatic complexity) into JavaScript according to 4.4 and 2.1.5 and the later with McCabe's imperative approach according to 2.1.4. What is noteworthy is that TFG has a higher cyclomatic complexity than IG with a value of 3.125 compared to 2.090. The cyclomatic complexity of TFG represents an increase of 49.5 % compared to IG. If we on the other hand compare IG with FG, IG has a higher cyclomatic complexity with a value of 2.090 compared to 1.136 which is a 83.98% higher cyclomatic complexity. Moreover we assume that the complexity difference of 49.5% between TFJSG and IG is mostly caused by the fact that those implementations are not related to each other in any way except for being an implementation of a directed graph program. They are not similar in terms of number of implemented functions, neither are responsibilities within the programs arranged similarly which make them a bit hard to compare. One thing to notice though, is that TFG consists of nearly 40 lines of code while the IG consists of around 100. While TFG is a smaller program than IG in terms of lines of code, TFG might have a significantly higher cyclomatic complexity since the functional programming style is much more compact. This increases complexity in terms of the density of logical operators, identifiers, non-identifiers and list-comprehension like code per line. Our approach of measuring cyclomatic complexity on functional JavaScript according to 4.4 reveals the program's complexity while McCabe's imperative approach deceptively shows an almost non-existent complexity.

5.3 Previous research

Van den Berg present numbers regarding his measurement, numbers which show that his cyclomatic complexity formula was sufficient for measuring on the functional language Miranda. The experts found that the code in fact were complex but the experts did not agree on the readability of the Miranda programs and Van den Berg suspects that it could mean that there is not an accepted standard on readability for Miranda as the one present for Pascal. Van den Berg also measured the effort rank and the rank on the cyclomatic complexity number between the two languages, which resulted in significant correlations. Van den Berg says that this indicates a consistent measurement of the syntactic complexity metrics developed in his study. He also shows that the correlation between the average expert rank and the cyclomatic complexity order for Pascal is equal to 0.58, and for Miranda is it equal to 0.56. Which displays two values that are close to each other, just like our results for the measurement on the Haskell and TFG implementations.

5.4 Research questions

With both our method executions and the results we have answered both of our questions. Even though the first research question might be somewhat trivial, this question needed to be verified in case of this study which we could not find in any previous research. The second research question is the essential question in this study.

7 Conclusion

In the case for RQ1, we can conclude that in fact McCabe's imperative approach for measuring cyclomatic complexity is insufficient when conducted on functional JavaScript. By measuring cyclomatic complexity for both FG and IG we can draw the conclusion that the cyclomatic complexity is consistently too low and misleading. The result is an average cyclomatic complexity of approximately 2 per function for the imperative implementation and an average of approximately 1 per function for the functional implementation. The functional implementation is similar to the imperative and therefore should get a higher value, closer to the imperative one.

To answer RQ2 we translated Van den Berg's approach according to 2.1.4 for measuring cyclomatic complexity on functional languages to be applicable on functional JavaScript according to 5.4. Measurements on functional JavaScript performed with our translated approach yield results fairer than McCabe's imperative approach since it produces results actually reflecting the complexity. The difference in cyclomatic complexity between HG and TFG is 0.375 and between TFG and IG there is a cyclomatic complexity of 1.035 hence we conclude that our approach of measuring cyclomatic complexity on functional JavaScript better reflects the complexity of the program.

To possibly get better results we could have considered implementing an alternative imperative graph program corresponding to TFG to increase accuracy and validity in comparing cyclomatic complexity between corresponding functional and imperative implementations respectively. We assume such a comparison would have led to smaller differences between the two.

Our adapted formula can also be applied and adapted on other multi-paradigm languages such as Python and Ruby to produce similar results.

7.1 Future Research

If we had more time to our disposal would we have wanted to use people to evaluate the same code that we applied the cyclomatic complexity measurement formula on.

In order to strengthen our results on RQ2, a good option would have been to conduct a study with a group of human participants to collect data on how they perceive the complexity of code under measurement. This would have helped us to validate if our formula of cyclomatic complexity is more

trustworthy by having humans reasoning and grading the complexity of code blocks.

In our discussion on validity in 3.3 it is mentioned that an imperative equivalent to the graph program translated from Haskell to JavaScript would be optimal for comparing and evaluating the cyclomatic complexity measured with our approach. It is also mentioned that a larger code base for measurements would add value to the results of this thesis with more algorithm functions translated from the Data.Haskell core package into the corresponding JavaScript implementation such as a *depth-first search*, a *breadth-first search* and a *transitive closure*. Therefore we suggest a replicated study improving those issues in order to optimize our approach for measuring cyclomatic complexity in functional JavaScript.

With a working formula, it would have been interesting and useful to make a tool like Jshint but to only measure cyclomatic complexity according to the formula described in 2.1.4 and 4.2. This tool could provide meaningful information in runtime to eliminate complex code and even oust bugs early on in the development.

With the adaptations made on the formula in 2.1.4 to fit JavaScript described in 4.4, we concluded that the formula worked to measure cyclomatic complexity on functional JavaScript is a strive in the right direction but since JavaScript is a language that can be written in either paradigm it is not enough to cover these cases. The formula will cover code that is written in the functional paradigm and McCabe's formula will cover code that are written in the imperative paradigm. Because of this, future work regarding measuring cyclomatic complexity in JavaScript is very interesting. To come up with a formula that cover the cases when code written in JavaScript are mixed in both paradigms, we suggest that it would be possible to solve those cases by defining a new formula that takes the necessary parts from both the formula described in 4.4 and McCabe's formula. By for example taking the part of the formula described in 2.1.5 regarding number of guards we cover the definition for counting possible paths through the code such as if-statements and switch-statements that is part of McCabe’s formula. If we stop counting the number of pattern identifiers as well as the unique pattern identifiers from the formula and replace them since they will probably be a problem when measuring on imperative JavaScript, we could still count the number of logical operators and the number of filters in list comprehensions since we doubt that those will be a problem for imperative. The problem lies as mentioned in the pattern complexity so the pattern complexity in list comprehensions also needs to be

replaced. We think that focus should be on the RHS, the LHS does not matter so much and will not significantly increase the cyclomatic complexity number.

References

[1] *Final Draft Standard ECMA-262 Edition 5.1, March 2011 (Rev. 6)*. 1st ed. Ecma International, 2016. Web. 26 Apr. 2016.

[2] "Reactive-Extensions/Rxjs". *GitHub*. N.p., 2016. Web. 23 Mar. 2016.

[3] "Folktale". *GitHub*. N.p., 2016. Web. 12 May 2016.

[4] "Ramda/Ramda". *GitHub*. N.p., 2016. Web. 12 May 2016.

[5] "Reactjs/Redux". *GitHub*. N.p., 2016. Web. 13 May 2016.

[6] "Cycle.Js". *GitHub*. N.p., 2016. Web. 13 May 2016.

[7] "Fantasyland/Fantasy-Land". *GitHub*. N.p., 2016. Web. 12 May 2016.

[8] Ryder, Christopher. "SOFTWARE MEASUREMENT FOR FUNCTIONAL PROGRAMMING"., University of Kent at Canterbury, 2004

[9] Van den Berg, Klaas. "Software Measurement And Functional Programming"., University of Twente, 1995

[10] "Lambda Expressions (The Java™ Tutorials > Learning The Java Language > Classes And Objects)". *Docs.oracle.com*. N.p., 2016. Web. 16 May 2016.

[11] "Functools — Higher Order Functions And Operations On Callable Objects — Python V3.0.1 Documentation". *Docs.python.org*. N.p., 2016. Web. 16 May 2016.

[12] Springer, G. & Friedman, D.P. (1990). Scheme and the Art of Programming. Cambridge, MA: MIT / New York: McGraw-Hill.

[13] Gyori, Alex et al. *Crossing The Gap From Imperative To Functional Programming Through Refactoring*. 2013. Print.

[14] *Proposed Ecmascript 4Th Edition – Language Overview*. 1st ed. Adobe Systems Inc., The Mozilla Foundation, Opera Software ASA, and others., 2016. Web. 4 May 2016.

[15] T. J. McCabe, "A Complexity Measure," in *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[16] H. D. Mills, "Mathematical foundations for structured programming," Federal System Division, IBM Corp., Gaithersburg, MD, FSC 72-6012, 1972.

Appendix A

A1

The functional JavaScript graph implementation is found at the Github repository link below in the folder graph.
https://github.com/drager/thesis-material/tree/master/functional-javascript

A2

The imperative JavaScript graph implementation is found at the Github repository link below in the folder graph.
https://github.com/drager/thesis-material/tree/master/imperative-javascript

A3

The Haskell graph implementation is found at the Github repository link below in the folder graph.
https://github.com/drager/thesis-material/tree/master/haskell

A4

The translated JavaScript graph implementation is found at the Github repository link below in the folder graph.
https://github.com/drager/thesis-material/tree/master/graph-haskell-translated