FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Writing Efficient JavaScript Programs: a Performance and Optimization Study

Filipe Miguel Leitão Ribeiro

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Monteiro de Oliveira Restivo

Supervisor: António Pedro Freitas Fortuna dos Santos

February 25, 2019

Writing Efficient JavaScript Programs: a Performance and Optimization Study

Filipe Miguel Leitão Ribeiro

Mestrado Integrado em Engenharia Informática e Computação

February 25, 2019

Resumo

O número de websites existentes tem aumentado a um ritmo maior que o aumento verificado no número de utilizadores da internet, o que leva a que haja uma grande competição entre os proprietários dos websites pelos seus utilizadores. A melhor forma de um website ganhar vantagem em relação aos seus concorrentes, é fornecendo uma user experience superior, o que pode ser atingido através do aumento da performance da sua aplicação.

Como a maioria dos websites atualmente ativos inclui JavaScript, o conhecimento de boas práticas e de como otimizat esta linguagem são de extrema importância. Existe um conjunto de regras, as Bentley Rules, que visa melhorar a performance de programas. No entanto, estas regras foram definidas antes da linguagem JavaScript ter sido criada, e a sua aplicabilidade a esta linguagem ainda não foi estudada.

A ideia para esta dissertação veio da Jscrambler, uma empresa cujo principal serviço é a ofuscação de aplicações JavaScript, assim como outras soluções de segurança para este tipo de aplicações. No entanto, técnicas de ofuscação têm geralmente um impacto negativo na performance dos programas, o que gera a necessidade de otimizar o desempenho dos mesmos.

Assim sendo, os principais objetivos desta dissertação são estudar as Bentley Rules e a sua aplicabilidade a JavaScript, verificar quais têm o maior impacto na performance de programas JavaScript e finalmente implementar uma transformação de código, tirando proveito de uma ferramenta de transformação de código JavaScript, a ferramenta Jscrambler.

Na primeira fase deste projeto, a aplicabilidade de cada regra foi estudada. 19 das 26 regras estudadas foram consideradas aplicáveis. Para estas regras, foram criadas test fixtures para medir o impacto de cada uma na performance de programas JavaScript. Quatro regras sobressaíram em termos de resultados obtidos, e elas são, em ordem decrescente de impacto, *Precompute Logical Functions*, *Data Structure Augmentation*, *Loop Unrolling*, *Store Precomputed Results*.

Na segunda fase do projeto, a regra *Store Precomputed Results* foi escolhida, de entre as quatro mencionadas, para ser traduzida numa transformação de código. Depois de o desenvolvimento ter sido concluído, foram realizados testes com um jogo JavaScript open source, e ficou demonstrado que esta transformação, embora com alguma interação humana, foi capaz de aumentar a performance do jogo em cerca de 7 %, o que pode ser a diferença entre uma boa ou má user experience.

Esta dissertação prova, então, que as Bentley Rules são aplicáveis à linguagem JavaScript, e que elas podem ser benéficas sempre que lhe são aplicadas. Além disso, prova, também, que é possível desenvolver uma transformação de código semi-automática para melhorar o desempenho de programas JavaScript.

Abstract

The increase in the number of websites has been increasing at a faster pace than that of the increase in internet users, which leads to a fiercer competition between website owners for users. The best way for a website to gain the upper-hand on a competitor is by providing superior user experience, and this can be achieved by having a more performing application.

As the majority of websites includes JavaScript, the knowledge of good practices and how to optimize this language are of utmost importance. There is a set of rules, the Bentley Rules, which is aimed at improving the performance of programs. However, they were defined before JavaScript was created, and their applicability to this language has not been studied.

This idea for this dissertation came from Jscrambler, a company whose service is the obfuscation of JavaScript applications, as well as other security solutions. However, obfuscation techniques usually have a negative impact on the programs' performance, which is why optimizing the code's performance is of high priority.

Therefore, the main goals of this dissertation are to study the Bentley Rules and their applicability to JavaScript, verify which have the biggest impact on this language's programs, and consequently implement a code transformation by taking advantage of an already developed JavaScript transformation tool, the Jscrambler tool.

In the first phase of this project, the applicability of each rule was studied. 19 out of the total 26 rules were considered to be applicable. For these rules, test fixtures were created to measure their impact on JavaScript code's performance. Four rules stood out with their results, and these rules are, in decrescent order of impact, *Precompute Logical Functions*, *Data Structure Augmentation*, *Loop Unrolling*, *Store Precomputed Results*.

In the second phase of the project, the *Store Precomputed Results* was chosen, out of the aforementioned rules, to be translated into a code transformation. After the development was concluded, tests were made with an open source JavaScript game, and it showed that this transformation, although with guidance, was able to increase the performance of the game around 7% which can be the difference between a good user experience or a bad one.

This dissertation thus proves that the Bentley Rules are applicable to the JavaScript language and that they can be beneficial whenever applied to it. Furthermore, it proves that it is possible to develop a semi-automatic code transformation to improve the performance of JavaScript programs.

Acknowledgments

I cannot consider this dissertation complete without, before, expressing my deepest gratitude towards some people who have been crucial to my development both personally and professionally, during the last few months — some during my entire life. And these people are, in chronological order:

My entire family, who has provided massive support, sometimes when even I did not know it was needed. A special word to my mother, Margarida, who has always fought and done everything in her power so I could have the brightest future possible, and my sister, who has helped me grow into who I am today.

Secondly, my friends, some of whom I don't even remember not being a part of my life. They've always provided me with laughs when I was in need of destressing, and incentivized me when I needed a push in the pursuit of my dreams.

Juliana, who was by my side since before I started this journey at FEUP. She always incited me to become a better and more ambitious professional, as well as a more responsible and caring person.

Everyone at FEUP, from the teachers to every colleague who has helped me in this long journey. A big 'thank you' to my supervisor, André Restivo, for his knowledge and availability to help me whenever I was in need.

And finally, last but not least, everyone at Jscrambler, who provided me with an amazing environment for the development of this dissertation. They're amazing professionals, as well as amazing people, from whom I've learned a lot during these months.

Filipe Ribeiro

"Before I learned to walk,
I dreamed about running, one day.
Now I walk, run and jump,
Not that special, dare I say."

Filipe M. Ribeiro

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This initial chapter has the purpose of giving the reader a better understanding of the context of this dissertation, the problem trying to be solved, the motivation behind it, and its goals. Section 1.1 describes the context which this dissertation is inserted into. Section 1.2 describes the problem being tackled by this dissertation and the causes of this problem.

Section 1.3 presents the motivation behind this project and functions as a link between the problem of this dissertation and its main goals, which are then presented in Section 1.4. Finally, as a means to guide the reader through the document, the structure of the document is thoroughly explained in the last section of this chapter, Section 1.5.

1.1 Context

The internet is in vogue, having more users with each passing year (Stats, 2017). This requires more web applications to be developed to attend to every user's needs and preferences. In web development, one of the most used programming languages used is JavaScript, as it is present in nearly every active website today (W3Techs).

With the use of different tools, platforms or frameworks in the development phase of a website, as well as the developers' own skill and preference, source code may be written in many different ways. Consequently, the same website, when developed in different ways, will have different performances.

Throughout this dissertation, the term *performance* will be used multiple times, and it refers to the execution time of programs. In that sense, the faster a program is to execute, the more performing it is considered to be.

Many website owners are concerned about their websites' performance and protection, and Jscrambler is a company that provides a product, which is homonymous to the company, that guarantees code and data protection, by means of code obfuscation, to the clients' websites (Jscrambler). Applying techniques such as these causes overhead in the application and that has a big

impact on the code's performance. Therefore, it is of utmost interest for the company to try and reduce the overhead by being able to apply code optimization techniques along with the obfuscation. The Bentley Rules are the starting point of this search for possible optimizations.

This dissertation is a consequence of this desire from the Jscrambler company, and the starting point in this search for code optimization was the *Bentley Rules*. They are a set of rules defined by Jon Lewis Bentley to help developers improve the efficiency of their programs, and are seen in more detail in Chapter 4.

1.2 A Performance Problem

A major concern in software development is that developers tend to care first about functionality, and later about the performance of the programs they are developing. This may be because of short deadlines, or simply lack of awareness for performance needs. Regardless of why this happens, it causes some of the programs developed to not be as performing as they could, or should.

The following section explains why user experience dictates the loyalty of users towards websites, and why users may opt for faster websites for the same purpose. Therefore, it is of utmost importance that websites are as performing as possible to maintain their users, and possibly attract new ones.

1.3 Need for Performance

The previous section mentions that the low performance on programs, specifically in websites, have a great influence in user-experience. However, it only scratches the surface on explaining why that is believed. In the year 2000 it was estimated that, for each active website, there were 24 users on average (Stats, 2017). The same measure in 2015 states that the number of users per website is around 15% that of 15 years prior. This, coupled with the fact that many businesses depend on the number of active users on their websites, leads to a fiercer competition between websites for users, whether they are active users or just occasional ones.

Owners of websites that are original, or provide a unique service might assume performance is not something they should bother about, as users may have no other reliable option if they want to use said service. However, this is not the case with the majority of websites. Therefore, website owners with many competitors should focus on improved performances, as the users' decision on which website to use may come down to which one is the most performing and provides the best user experience. The reasoning behind this comes from the fact that, for several users, a delay of just a few seconds or even less than a second can, sometimes, cause the user to lose a business opportunity or other type of opportunity available from websites. It is estimated that a staggering 53% of users abandon websites that take more than 3 seconds to load (Shellhammer). Moreover, real-time web applications such as games, betting, or stock market applications, require high performance, as any delay could ruin the experience for its users.

Low performance generates high computational costs, which is undesirable for any user or any owner of a website. The expression *Computational Costs* is referent to the amount of resources used to execute the programs. Some machines may have a hard time executing very demanding programs, affecting user experience as well. As a further matter, code obfuscation done by Jscrambler's product has a damaging effect on performance. This is an incentive for the company to look for ways to optimize the code, so the effects of obfuscation do less damage to the user experience of their clients' product users.

1.4 Main Goals

The main goal of any dissertation is to enrich the field or area in which it is focused on. In this sense, this dissertation does not go astray from this rule, as the hopes are that it enriches the field of *Software Engineering* with the following contributions:

1. **Study the Bentley Rules and their applicability to the JavaScript language.** The inspiration for this dissertation was the set of rules known as Bentley Rules, and the first goal is to study if each individual rule is applicable to the JavaScript language. If a rule is deemed to be applicable, then its impact on the aforementioned language will also be studied, with the aid of tests performed in different JavaScript engines.

2. **Apply the studied rules to JavaScript programs using a code optimization tool.** This can be achieved by taking advantage of Jscrambler's obfuscation tool, which is capable of analyzing programs, and transform them according to specified patterns. This will be done for the rules that are deemed to be the most advantageous, according to the results of the first main goal.

1.5 Document Outline

This document contains six total chapters and is structured in the ensuing manner:

Chapter 1, "Introduction", which is the present chapter, gives an introduction to the topic of this dissertation. It includes the context, the problem, motivation and the expected contributions.

Chapter 2, "Background & State of the Art", contains all the theoretical foundations supporting this dissertation and required for the reader to fully understand everything being discussed throughout the document, and references what is being done in the same field of work. Different solutions to the same problem are reviewed as well.

Chapter 3, "Performance Problem", is where the problem to be solved is stated and explained in a more extensive manner, after being briefly described in Section 1.2.

Chapter 4 and Chapter 5 describe the steps taken into the achievement of the goals described in Section 1.4, and the results obtained in each part of this dissertation.

Chapter 6, "Conclusions and Future Work", contains the author's insight on the results obtained and points to future work on this subject.

Chapter 2

Background and State of the Art

This chapter contains the theory required to both enable the reader to understand and support the methodologies to be implemented.

Section 2.1 provides the reader with a better understanding of Source-Code Analysis. Section 2.4 provides the reader with a better understanding of Source-Code Optimization, the different techniques used

Each section explains the importance of the subject in the context of this dissertation.

2.1 Source Code Analysis

Source-code analysis has a crucial role in programming today. It is mostly used as means to an end, as the possibilities after it has been performed are immense. David Binkley makes a very compelling case regarding this subject (Binkley, 2007):

> "... source-code analysis is a means to an end. Under this view, the end is of paramount importance. It first must be defined. Only then can techniques for analyzing the source code in order to achieve this end be considered."

He then continues by mentioning the other cases in which source-code analysis is used, by saying:

> "The other end of the spectrum is captured by the saying "if you build it, they will come." Those who hold this perspective believe that a tool capable of extracting generally useful information from the source code will find application. One advantage of this approach occurs when the information developed finds unforeseen application."

This dissertation stands in the first case described, as the ultimate goal is the optimization of source-code, and to achieve that, the analysis of said source-code is essential.

In this section, two different types of source-code analysis will be explored. The first, in Subsection 2.1.1, is Dynamic Analysis, and the second, in Subsection 2.1.2, is Static Analysis. For each of these types, some techniques will be detailed in the respective subsection.

2.1.1 Dynamic Analysis

This type of code analysis is performed at runtime, i.e. when the program is executing. It is useful to better understand the program's behavior during execution Ball (1999), which is an important trait to detect unexpected abnormal behavior, such as programming mistakes, bugs or device failure, or even to detect the inability of the program to achieve the desired goal(s).

> "Dynamic program instrumentation and analysis enables many applications including intrusion detection and prevention, bug discovery and profiling. " Chow et al. (2008)

Out of these mentioned applications of dynamic analysis, the most relevant in the context of this dissertation is runtime profiling. This type of dynamic program analysis is able to gather very useful information for a future optimization of the program Hauswirth and Chilimbi (2004):

1. **Time complexity.** Profiling is able to detect how much time is spent in the execution of the complete application, a particular function of a specific block of code instructions. By detecting this, it is possible to conclude what are the program's bottlenecks, which are the parts of the program that vastly hinder the program's performance. With this information, the user is able to know what parts of the program need to be improved more urgently to increase the performance.

2. **Code coverage.** Profiling also allows an understanding of what parts of the program are executed more often or even those that are not executed at all. This is important to detect dead code, which is code that is never executed. This code can almost always be removed, reducing the program's size.

3. **Functions usage.** Profiling is not only able to detect how often a function is called, and how long it takes to be executed, but it also detects what type of arguments it receives, their values, and what the function returns in every execution.

2.1.2 Static Analysis

Unlike dynamic code analysis, this type of code analysis is performed without executing the program. This has the advantage of being faster than its counterpart, as the need for executing is non-existent, however, the information obtained if merely compile-time data, limiting the full understanding of runtime behavior of the program. But several techniques exist to improve this type of code analysis and make its results come close to those achieved in dynamic code analysis. Some of these techniques are:

1. **Abstract interpretation** (Cousot and Cousot, 1992) is a method used for providing an idea of how a program behaves during runtime. By means of abstraction, the goal is to ensure the semantics of a given program is in accordance with the expected behavior.

2. **Data flow analysis** is a method used for a better understanding of the variables used in the program and the parts of the code to which those variable might be propagated to. By doing this, it is easy to understand which variables are not used during the execution of the program, for example.

3. **Control flow analysis** is a method used to gather information on the impact of each instruction of a program, thus generating an understanding of how a program executes in terms of which functions are called, and what parts of the code will or won't be executed.

4. **Hoare logic** (Floyd, 1967) has the main objective of reasoning about the correctness, equivalence and termination of programs, by means of a formal system.

5. **Model checking** (Baier and Katoen, 2008) is used for verifying every possible state in which the program can be in, so then there can be certainty whether a program meets certain specifications.

6. **Symbolic Execution** (Baldoni et al., 2018) is a technique used for understanding what the program's behavior is like for different inputs. This is useful for finding possible future causes for an execution failure, for example.

2.2 Optimization of Programs

The optimization of programs can occur at different levels (Bentley, 1982). Depending on the time of the optimization, at some of the levels, it becomes increasingly harder to implement. At a higher level, optimization should be made as soon as possible, since the lower levels will depend on the higher ones. Changes in higher levels of the program may cause the remainder of the program to be altered as well.

The different levels of optimization are:

2.2.1 Design Level

Design level optimization (Samuel and Kovalan, 2016) is the highest level of optimization possible. Modifications to the design of a program might mean that the entire code of the program has to be rewritten. However, small improvements in the architectural design of a program may greatly increase performance.

It is believed that design modifications should be made before the development phase has begun, to prevent the entire remodeling of the program's code, which can be very costly. Because of this, it is advisable that developers pay close attention to the design to be implemented, or possibly follow design patterns applicable to the specific scenario.

2.2.2 Algorithm and Data Structure

The choice of a data structure or algorithm can have a lesser impact than design choice, but still a great impact nonetheless on a program's performance (Wirth).

The general idea behind algorithm and data structure optimization is about balancing both size and time performing operations. For example, the addition of information to a data structure could mean less operation when trying to calculate it, but adding too much information to a data structure may cause reading it to be too much time-consuming.

Some of these optimizations are included in the Bentley Rules.

2.2.3 Source Code

There are some ways to achieve optimization at the source code level, and this is the level of optimization that will be studied in more detail in the next section, Section 2.4. Source code is the implementation of the choices made in the design, and algorithm and data structure levels. However, it can still be very influential towards the performance of the program.

Optimization at this level includes modifying loops, eliminating dead code, among other modifications of the code that has been written by the developers of the program (Šimunić et al., 2000). Several source code optimizations are included in the Bentley Rules, and will be studied more thoroughly.

2.2.4 Build

Below the source code level is the build level. The building of a program encompasses processes such as automated tests generation, which can be either helpful for debugging, or unhelpful in the sense that they are time-consuming.

Optimization at this level does not usually have a staggering impact on the program's performance.

2.2.5 Compilation

During compilation, modifications to the source code can still be made. The predicted solution to be achieved by this dissertation is done on this level, as the optimization tool will function as a compiler.

Because of this, the source code optimizations mentioned in the respective subsection can also be applied to this level.

2.2.6 Assembly

The assembly level is the lowest level where optimization can occur. At the moment, it is nearly impossible to optimize code in assembly more than what the existing compilers do. Instead, optimization is done at higher levels.

2.2.7 Run Time

Some compilers have the ability to perform optimization at runtime.

However, this type of optimization is usually only required when dealing with code that is capable of reflection, that is, to modify its structure, generate more code or alter its behavior while being executed.

2.3 Bentley Rules

This set of rules written by Jon Louis Bentley was the inspiration for this dissertation, as mentioned in Chapter 1. The purpose of these rules is to give programmers some guidelines on how to write their code with the goal of achieving superior programs in terms of performance.

In Subsections 2.3.1 and 2.3.2, two different types of *Data Structure Modifications* will be introduced. The first section describes *Space-for-Time* rules, and the second section describes *Time-for-Space* rules.

In Subsections 2.3.3, 2.3.4, 2.3.5 and 2.3.6, four different types of *Code Modifications* will be introduced. The first of these subsections describes techniques for *Loop* modification and optimization. The next subsection describes techniques for *Logic* modification and optimization. Then, the following subsections describes rules for *Procedure Design* optimization, and the last of these subsections describes rules for the optimization of expressions.

2.3.1 Space-for-Time Rules

Space-for-Time rules have the sole purpose of reducing the time of execution of a program. However, in exchange, the size of the code may be increased. Bentley specified four different *Space-for-Time* rules:

1. **Data Structure Augmentation** — The goal of this rule is to modify the structure of the data, creating additional information, to prevent the operations required to calculate said data on the program's part, therefore reducing the execution time.

   As an example, in a data structure which contains information about people, if the information contains the birth date, and the zodiac sign has to be calculated from this date, the rule suggests adding an additional field to the data structure with the zodiac sign of each person. This would make accessing it easier, as the calculations are no longer necessary.

2. **Store Precomputed Results** — As the name of this rule suggests, the aim is to store the results of functions, so that they can be used subsequently, whenever that function is called again.

   By doing so, repeated calls to a function do not require the execution of said function, instead just looking up the result in the data structure where the results are stored.

   An example of the application of this rule is the call to a function which returns the Fibonacci numbers. Being a recursive function, if a large number was stored, and an even larger one

was requested, there would be no need to recompute all the Fibonacci number until the first one, instead acting as the starting point for that execution, and preventing unnecessary computations.

3. **Caching** — This rule is based on the idea that data that is accessed more often should be the cheapest to access. For this to be true, this data should be cached for easier access in future uses.

4. **Lazy Evaluation** — This technique aims at avoiding the computation of instructions that are not required at a given time, instead executing them only when it is essential.

   A different application of this rule is memoization, which is a technique to store results, to avoid future re-computations. However, it only works with pure functions.

2.3.2 Time-for-Space Rules

Time-for-Space rules have the sole purpose of reducing the size of the code of a program. However, in exchange, the time of execution may be increased. Bentley specified two different *Time-for-Space* rules:

1. **Packing** — This rule aims at reducing the storage costs, by increasing the time consumed in storing and retrieving data. An approach to apply this technique is modifying the format of the data that we want to store.

   An example would be the opposite of that given in the *Data Structure Augmentation* rule. If a database contains information related to users, and it contains both the date of birth and the zodiac sign of a person, then the zodiac sign attribute should be removed, and whenever it was required during the execution of the program, it would be calculated by using the date of birth.

2. **Interpreters** — By using interpreters, it is possible to reduce the space of the program, since interpreters have the ability to compact sequences of operations.

2.3.3 Loop Rules

Loop Rules have the purpose of attempting to improve the performance of the code by modifying, or removing the loops detected in the program.

Loops are, in most cases, the part of the code that expends more time when executing a program. Therefore, they require particular attention as optimization is concerned. Bentley specified five different loop optimization rules:

1. **Code Motion Out of Loops** — This rule focuses on switching part of the code inside a loop to outside of it. This prevents the same operation to be executed every time the loop is executed, instead just being executed once, before the loop starts. This can only be done, however, if that part of the code is an *invariant*[1].

[1] An invariant is a variable inside the loop, whose value does not depend on the variable of the loop.

An example of the application of this rule would be the removal of a variable declaration that is going to be used inside the loop. If it is not going to be changed between iterations, then it can be defined before the loop starts.

2. **Combining Tests** — This rule focuses on removing any unnecessary tests from a loop. To do so, it suggests finding ways to combine exit conditions, consequently only performing a single verification.

   An example of the application of this rule would be the search of a variable in an array. Instead of performing the verification for each element of the array, and constantly verifying both if the iterator is larger than the size of the array, and if the element of the array is equal to the desired variable, the rule would suggest adding the variable to the array, and only performing the verification related to each element being equal to the variable. In this case, the variable would always be found on the array, only later performing the verification on whether the element found was the one inserted beforehand. The loop has one less test in every iteration, making it more performant.

3. **Loop Unrolling** — As the name indicates, this rule aims at unrolling unnecessary loops, consequently removing the verification of the end of the loop.

   Unrolling a loop consists of taking a loop with n iterations, and replacing it with the code inside its body, repeating it n times.

4. **Transfer-Driven Loop Unrolling** — This rule states that whenever there is a part of the code inside a loop with only trivial assignments, it is possible to duplicate the loop, but with a different use of the variables, removing those trivial assignments as a consequence.

5. **Unconditional Branch Removal** — This rule aims at modifying the fast loops that contain an unconditional branch at the end. Instead, it is preferable to have a conditional branch in its place.

6. **Loop Fusion** — This rule is suited for programs that have more than one loop operating on the same set of variables. It suggests fusing the loops, to prevent additional computations.

   An example of the application of this rule would be having two loops, one which accesses a data structure containing people's personal information, and for each element in it, accesses the name of each person, and a different loop which accesses the same structure and gets the age of each person. Instead, having a single loop which does both those tasks would be more performant.

2.3.4 Logic Rules

This group of rules has the purpose of optimizing the parts of the code related to the logic of programs.

It is expected that when these rules are applied in a program, the code becomes harder to understand by anyone who tries to read it as they trade readability for efficiency. Bentley specified five different *Logic* rules:

1. **Exploit Algebraic Identities** — This rule aims at replacing costly expressions in the code, for equivalent, but less costly expressions.

   A relevant example would be replacing the expression $x^2 < 100$ with $-10 < x < 10$. Doing so would reduce the cost since it would no longer be necessary to calculate the squared value of x.

   Another example of the application of this rule would be removing the last comparison in a ternary comparison[2]. In this case, if the first two possibilities were not true, the third one is always true. Therefore, removing it reduces the cost of the program.

2. **Short-Circuit Monotone Functions** — This rule prevents monotone functions from keeping the execution going whenever the outcome has already been decided.

   An example of this would be a function which goes through an array of arrays, where each element is relative to an employee of a company, and contains their salary, bonuses and other types of income, and that function intends to return the employee which received the least of all employees. If the current element being compared has already surpassed the current minimum's total whenever the first value is accessed, then it is not required to verify the other values on the same element, as its total will never be lower than the minimum's, therefore preventing unnecessary computations.

3. **Reorder Tests** — This rule states that tests that are more likely to pass should come before the hardest ones.

   As an example, in a function that receives a list of grades, in percentage, and associates each percentage to a nominal scale (e.g. 0-39 = bad, 40-69 = average, 70-100 = good), then the verification relative to the most usual interval for the scales given should be the first in the code. In the case the numbers received are random, the first verification, in this case, would be relative to the interval which covers more of the possible numbers.

4. **Precompute Logical Functions** — Whenever there is a logical function with a finite domain, in most cases it is more effective to precompute the function for every argument, and replace it with a table. This way, instead of executing the entire function for a given argument, it would just be necessary to perform a lookup on the generated table.

   An example of the application of this rule would be to take a function that returns the Fibonacci's number of a given number, *num*. In this case, the computational costs of calling this function for a large number may be too high as the function will be called recursively *num - 1* times. Hence, the solution of calculating every Fibonacci number, and storing the result before execution would prevent these computations during execution.

[2]Ternary comparisons are the cases where the three possibilities >, < and = are tested for

5. **Control Variable Elimination** — This rule states that whenever there is a control boolean variable, we can replace it in the code by an *if-then-else* statement, where instead of comparing the variable, we compare the expression or expressions that define the value of the control boolean variable.

   As an example, whenever a loop is trying to find a name in an array, instead of having a variable which controls when the name has been found, and whose value needs to be verified for every element searched in the array, have the function evaluate the value of the expressions which would turn that variable true.

2.3.5 Procedure Design Rules

Procedure Design Rules do not focus on modifying the code in a way that it functions in a different manner. Instead, these rules modify just the structure of the program. Bentley specified five different *Procedure Design* rules:

1. **Collapse Procedure Hierarchies** — This rule states that, instead of having procedures call other procedures, replacing those procedure calls with the body of the procedures themselves would save some time. This causes instructions to be removed from the code.

   A very simple example of the application of this rule would be a function that, at a specific point, calls a different function that returns the sum of two numbers. That function call should, instead, be replaced by the instruction that performs the sum of the two numbers.

2. **Exploit Common Cases** — This rule aims at handling the common cases more efficiently. The implementation of this rule is achieved by having a procedure that performs correctly in every case, as well as having a second procedure that performs more efficiently, however only for a few, special cases.

   An example on the application of this rule is having a function which returns the Fibonacci equivalent for a given number. If the developer has the information that the function is called frequently for a specific set of numbers, then it can have the Fibonacci's numbers specific to those stored, and then it can return them without having to perform the calculations.

3. **Use Coroutines** — This rule states that whenever an algorithm has subroutines linked by temporary files, they can be run as coroutines and communicate, therefore preventing the need of the temporary files.

   An example of the application of this rule is when a function writes into a file, and the only use for that file is to be read by a consecutive function, then the writing and reading of said file can be avoided by fusing the two functions.

4. **Transform Recursive Procedures** — The run time of procedures that perform recursion can be very large, but there is a large number of techniques to improve the efficiency of these types of procedures. Among them, the solution of replacing a procedure's call to itself as its last action with a *goto* command.

5. **Use Parallelism** — This rule aims at taking full advantage of the hardware used to run the program, more concretely taking advantage of multiprocessor architectures.

   An example of its application is having a thread calculate all the Fibonacci sequence numbers for the even number indexes, and have a different thread perform the same action, instead for the odd number indexes. This should reduce the time consumption by about half.

2.3.6 Expression Rules

The last set of rules mentioned by Jon Bentley are the *Expression Rules* which focus on reducing the cost related to the evaluation of expressions in the code. Four new, different rules have been mentioned:

1. **Compile-Time Initialization** — As the name suggests, this rule aims at initializing every possible variable before runtime, to save the respective time. As such, every variable whose value is already known, and that remains unaltered throughout the execution of the program, can be replaced by its value in every future expression it is used in.

2. **Eliminate Common Subexpressions** — Whenever there is an evaluation of an expression that is made more than once throughout the program, and between any of them there is no change made to any of the variables of the expression, the result can be stored to be used the next time the same expression is evaluated.

   As an example, if in the body of the program, it is evaluated whether a variable has a value different than null, and it is detected that nothing in the code between those evaluations altered the value of said variable to null, then the result of the first evaluation should be stored and used, replacing the second evaluation, since it will always be the same result.

3. **Pairing Computation** — This rule states that whenever there are two expressions evaluated together in more than one occasion, they should be put together in a new procedure, if possible.

   An example of this situation is when a program tries to access the x coordinate of an object, and it usually accesses the y coordinate as well. Therefore, the best option would be to have a routine that returns both coordinates, instead of having two different routines, one for each of the variables.

4. **Exploit Word Parallelism** — This rule states that the underlying computer architecture's full width of words should be used when evaluating values in the binary notation. Exploiting parallelism in cases like this should increase performance.

2.4 Source Code Optimization

This section focuses on describing some source code optimization techniques. The Bentley Rules, described in the last section include many optimizations of this type. Therefore, in this section, two types of source code optimization techniques are described. These two techniques are not included in the Bentley Rules:

1. **Minification**. This is the act of removing every, or a large number of unnecessary characters from the source code of a program (Souders, 2008). These characters can be spaces, tabs, line breaks, parenthesis, among others. This process of character elimination causes the program to be *minified*, in other words, smaller in size, which generates both a gain in performance, and decreased download time.

   Minification, however, has the disadvantage of making the code harder to read for humans. Human developers are generally accustomed to inserting line breaks, tabs, and even some extra, unnecessary parenthesis to increase code readability and flow understanding. When applying minification, these peculiarities are undone.

2. **Refactoring.** Martin Fowler, in his book, which made the process of refactoring popular explains refactoring in a very clear way:

   > "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written" (Fowler et al., 1999)

   Besides minimizing the chances of introducing bugs, refactoring helps developers improve the design of the software in question, making it easier in the process, which in turn can also help find some previously unidentified ones. By having an improved software design, it helps developing faster.

   But, is refactoring really code optimization? Martin Fowler answers:

   > "I see optimization and refactoring as two separate things, even though they often use the same transformations, and a particular transformation you do to your program may be both." (Fowler, 2004)

   He, then, continues explaining that although some of the transformations to the code are identical, the idea behind them is different. However, when doing refactoring, code increases in performance, so it is the author's belief that it should be considered a type of source code optimization.

2.5 Code Obfuscation

The knowledge of code obfuscation is very important in the context of this dissertation. Code obfuscation is the act of modifying the code so that it becomes ineligible. Code ineligibility is a means of security, as an attempted tampering with a website's source-code may be rendered useless if the attacker has no understanding of the program's logic (Collberg et al., 2003).

Although obfuscation is often used as a security measure, it can also be used by malware, to prevent being detected by any threat detection software (You and Yim, 2010).

Another use for obfuscation is in the protection of a program's intellectual property, in this case, preventing the source code being copied by competitors.

The program, despite its source code being modified, maintains the same logic and is functionally equal to the original.

Many techniques exist to obfuscate a program's source code, such as (CERT-UK, 2014):

1. **Minification.** Although mentioned as a source code optimization in Section 2.4, as it reduces the size of the program, it can also be considered a type of obfuscation of the code. The removal of any blank spaces and line breaks makes the code harder to be read, and consequently harder to understand its logic.

2. **Names Modification.** In a well-developed program, variables and functions have names that describe their purpose. This makes code easier to understand, and by renaming them to senseless names will create confusion to whoever is trying to read them. This, however does not work with automated deobfuscators, which do not associate a name with the variable or function's utility.

3. **Expression Replacement.** Modifying some expressions in the code to equivalent but more complex ones is a different way of making the code harder to read and understand.

4. **Dead Code Insertion.** Adding code that will have no impact in the logic of a program is one technique that will create confusion to an attacker.

5. **Control Flow Flattening.** This technique encapsulates the entirety of the code in just a single function, with each possible path inside a switch statement. That function is then called recursively, with different parameters depending on the part of the code it is to execute after. This generates confusion to anyone trying to reverse engineer it.

The application of obfuscation to a program has many advantages, as mentioned. However, it has a major drawback, which is the program's decrease in performance. This trade-off between performance and readability is something developers or website owners have to be careful about.

This knowledge of obfuscation is relevant in the context of this dissertation, as the Jscrambler tool has the main purpose of obfuscating web applications to protect their code, and privacy, which in turn creates a need for performance optimization as well.

2.6 Pure Functions

Pure functions are functions which always return the same result for the same set of arguments. To guarantee that a function is pure, some things need to be considered:

1. **The function must receive, at least, one argument.** Without receiving arguments, the function is either not pure, or it always returns the same value, which would not make much sense in programming.

2. **The function must always contain a return statement.** Not only does a function need to contain at least one return statement, but it also needs to contain one return statement in every possible "route" taken inside its body.

3. **The function's return value must not depend on external variables, or calls to impure functions.** It is true that if the function depends on an external variable, not changed throughout the execution of the program, the return value will not change as well. However, this is a case that is impossible to predict with static analysis of the code. Therefore, once the Jscrambler product is only capable of performing static analysis, whenever an external variable is used, the function in question is considered impure. Calling functions that generate random numbers are an example of how the behavior of the function may vary for the same arguments, making it impure in the process.

4. **The function may not produce side effects on the program.** Undesirable side effects include the output of information to the user, the use of randomly generated numbers, accessing the current date, and/or time, manipulating the Document Object Model, or making HTTP requests. All of these have the potential to make the result of a function in different executions, for the same arguments.

This knowledge on pure functions will be important in Chapter 5, as the code transformation implemented, and there described, is only applicable to this type of functions.

2.7 Support Technologies

During the development of this dissertation, some technologies were used that require previous explanation on their means of operation or the goals they help to achieve.

In the first subsection, Esprima, an ECMAScript parser, is presented. This technology is used by the tool presented in the second subsection, the Jscrambler tool. This tool is described, with an in-depth explanation of its features.

The third and fourth subsections present JSPerf and Browserstack, respectively. These two technologies were used in the validation part of this dissertation.

2.7.1 Esprima

As can be read in the front page of the Esprima website, Hidayat:

> "Esprima is a high performance, standard-compliant ECMAScript parser written in ECMAScript (also popularly known as JavaScript)."

This technology has the ability to read a string of valid JavaScript code, and transform it into an Abstract Syntax Tree (AST), which is a tree that represents the structure of the source code. This transformation from source code into ASTs makes the analysis of the code and consequent transformations much easier and more intuitive to perform.

The following JavaScript code excerpt shows a very simple program, to be transformed by Esprima:

```
1 function sum(a, b) {
2     return a + b;
3 }
4
5 sum(1, 2);
```

Listing 2.1: Simple JavaScript program to be transformed into an AST

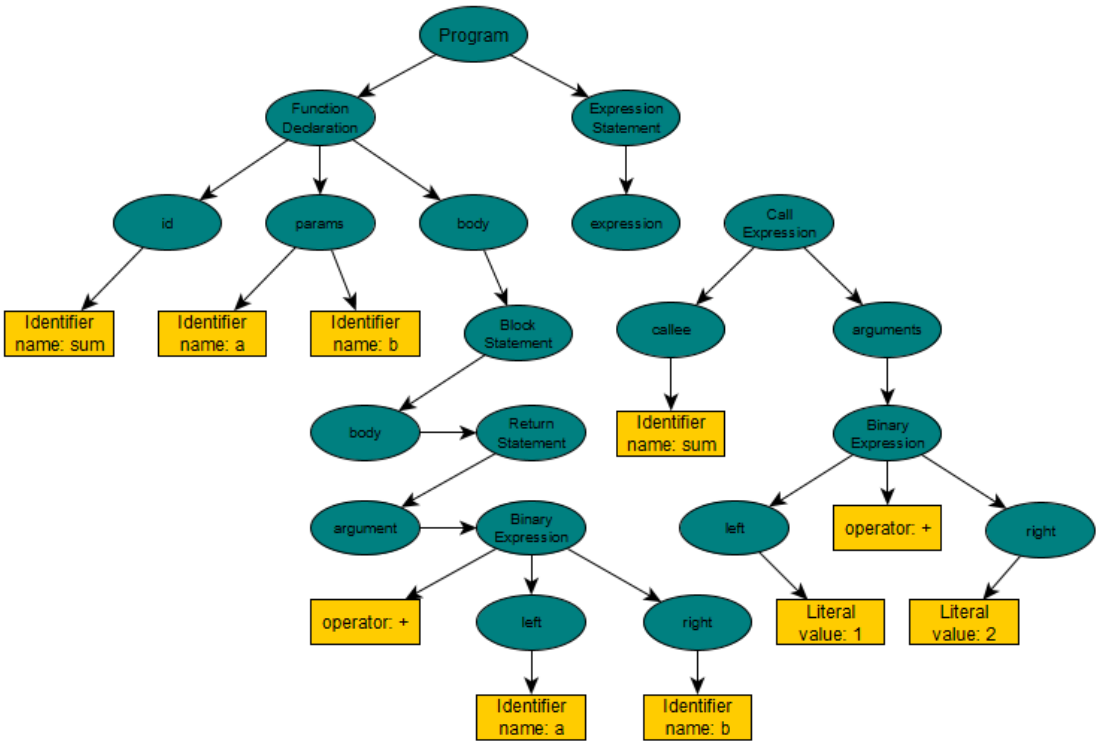

Figure 2.1: AST resultant of an Esprima transformation

Figure 2.1 represents the resulting AST from the parsing of the JavaScript code with Esprima.

2.7.2 Jscrambler

The Jscrambler tool has the purpose of protecting JavaScript web applications from attacks, as well as protect their data. To do so, it applies optimization and obfuscation techniques to JavaScript-based code.

At the moment, the optimization techniques this tool uses include:

Assertion Removal, which is a technique that aims at removing any assertion in the code. Assertions are statements that will always return true at their point in the code.

Constant Folding, which is a technique aimed at evaluating expressions at compile time, instead of run time. If the program, before executing, already has the means to evaluate an expression, it is evaluated before the execution, to avoid unnecessary computations.

Dead Code Elimination, which is a technique that removes every part of the code that is never executed.

Debug Code Elimination, which is the removal of any code whose purpose is only executed in the debugging process.

Duplicate Literals Removal, which aims at removing any repeated literals in the code, by assigning them to a variable and using the variable in their place.

Although the tool applies optimization techniques, it also applies many obfuscation techniques to protect the code, which causes the programs to lose performance. Therefore, more, and more impactful optimization techniques are required to counter the effect of the obfuscation ones.

Before applying any transformations to an application, some insight is gathered of the way its executions occur, however the tool itself can only perform a static analysis of the code, more specifically data flow analysis and control flow analysis, which were described in Section 2.1.

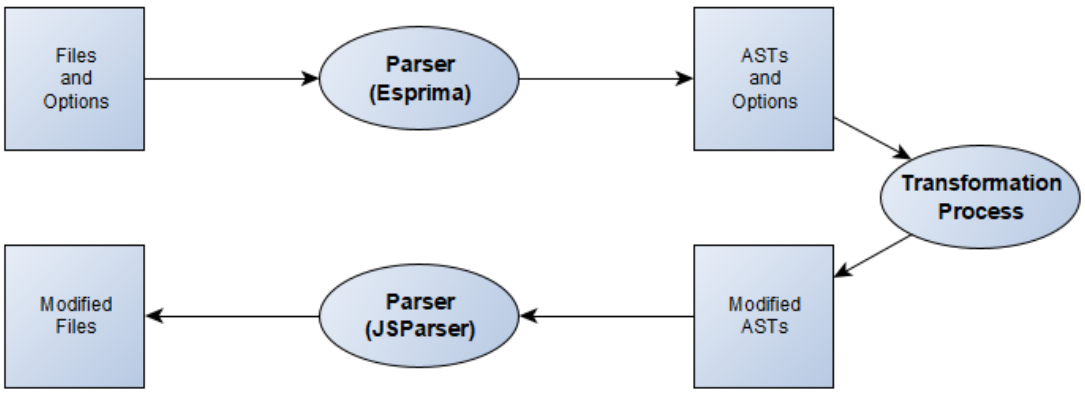

Figure 2.2: Jscrambler's data flow for a transformation

Figure 2.2 shows a simplified data flow for a transformation, which follows the ensuing steps:

1. **Receive the options and files to transform.** The first step for a transformation is to receive the files which are to be transformed, and options from the user. These options may be what

transformations to apply to each part of the project, or not to apply, or other options related to specific transformations.

2. **Parse the code into ASTs.** For each file received an AST is generated by using Esprima. A file is not parsed into an AST if it is specified in the options that it shouldn't be modified.

3. **Modify the ASTs.** For each AST, the Jscrambler tool will perform the transformations required according to the options received. The transformation process is detailed in Chapter 5.

4. **Generate the modified source code.** After the ASTs are modified, they are parsed into the modified source code. This is accomplished by a parser developed by Jscrambler.

2.7.3 JSPerf

JSPerf is a tool that helps with testing the performance of pure JavaScript code. It executes the tests specified by the user, and generates a small report on them.

By default, this tool runs each test at least 5 times, and for at least 5 seconds. This means that faster code being tested may run hundreds of times, while slower ones may execute only 5 times. However, it is possible to modify this, and have the tool execute every test the same number of times, for more statistically relevant results.

To generate the reports of each test, the tool gathers the time of execution of each sample, then combines them to calculate the operations per second for each test, as well as a relative margin of error, for the confidence interval of 95%.

In the end of executing each test, it shows the results, and it highlights the test which achieved the best results.

This tool was very useful in the development of this dissertation, as it was essential in the evaluation of the performance gain of each Bentley Rule, when applied to the JavaScript language.

2.7.4 Browserstack

This tool is a web testing platform that is very useful in cross-browser test automation, as it gives users access to hundreds of different browser versions, in different operating systems (OS), as well as different devices (Kaalra and Gowthaman, 2014). It allows the easy testing of websites or mobile applications in the required environment. This tool works by connecting the user's machine with a server, which performs all the actions, via RFB protocol.

Browserstack allows for both live testing, and automated testing. In live testing, a user is capable of opening the desired device, running the desired OS, and open the desired browser, and then use it at will. In automated testing, a user can develop the automated tests, with Selenium, and then perform the required test. However, some of the browsers do not support Selenium automated tests, as is the case with UC Browser. In that case, any test required, would have to be performed in live mode.

This tool was very useful in the development of this dissertation, because, along with JSPerf, it helped in the evaluation of the performance changes for each Bentley Rule in different JavaScript engines.

2.8 JavaScript Optimization Tools

This section has the purpose of providing some insight into what other solutions exist that try to solve the same problem, the low performance of JavaScript programs. The different approaches of each solution will be described.

Subsection 2.8.1 focuses on a tool called *Closure Compiler*, while Subsection 2.8.2 focuses on a tool called *Prepack*.

2.8.1 Closure Compiler

The *Closure Compiler* tool is a JavaScript optimization and minification tool developed by Google (Google).

The tool was experimented with to better understand what types of optimization it is capable of. The types of optimization tested were the ones mentioned in Section 2.3:

1. **Time-for-space and space-for-time rules.** It was not possible to understand whether the *Closure Compiler* is capable of applying these types of rules to a JavaScript program or it still needs upgrading in those cases.

   As mentioned in their respective sections, these rules are intended for modifying not the code itself, but the data structures behind a program.

2. **Loop rules.** The tool left much to be desired in terms of loop optimization, as it showed to be capable of applying *Code Motion Out of Loops*, but was not able to unroll a simple loop, as the one shown on code excerpt **??**, not even when the loop was required to run only three times. It was also unable to apply the *Combining Tests* or the *Loop Fusion* rules.

3. **Logic rules.** Very much like the *Loop* rules, this tool does not deliver in the *Logic Rules* context. From the experimentation, it was clear that the tool does not have the ability to perform code modifications regarding any of the five different specified rules.

4. **Procedure design rules.** The tool seems capable of performing the *Inline Expansion* technique.

5. **Expression rules.** The tool is capable of performing the *Constant Propagation* technique.

6. **Other types of optimization.** This tool is great at *Dead Code Elimination*, which, as the name suggests, is a technique used to remove parts of the code that have no effect on the result of the program. Such cases include variables that are never used, or unreachable parts of the code.

It also focuses on *Code Minification*, which is a technique that focuses on removing unnecessary spaces, line breaks, comments and other unnecessary characters in the code. This allows the program to be smaller, which saves time, increasing performance.

2.8.2 Prepack

Like the *Closure Compiler*, Prepack is a tool for JavaScript optimization purposes. It was and is still being developed by Facebook. It uses *Abstract Syntax Trees* for code analysis and transformation.

The tool was experimented with for a better understanding on the types of optimization it is capable of performing:

1. **Time-for-space and space-for-time rules.** Very much like in the case of the last tool mentioned, it was not possible to understand whether *Prepack* is capable of applying these types of rules to a JavaScript program or it still needs upgrading in those cases.

   As mentioned in their respective sections, these rules are intended for modifying not the code itself, but the data structures behind a program.

2. **Loop rules.** The tool was able to apply the *Loop Unrolling* rule. However, with loops with more than 40000 iterations, the tool stopped working, returning an error. Also, for loops of this size, it still unrolled them, which does not help the size of the code.

   It does not seem able to perform *Loop Fusion*, as instead of fusing the two loops, it instead unrolls both loops. The same is verified for the remainder of the *Loop Rules*.

3. **Logic rules.** From the tests made, it was concluded that the tool is not capable of applying the *Exploit Algebraic Identities*, the *Reorder Tests*, nor the *Control Variable Elimination* rules.

4. **Procedure design rules.** The tool does not seem capable of performing the *Inline Expansion* technique. It was not possible to conclude anything about the other *Procedure Design* rules.

5. **Expression rules.** From testing, the tool is capable of performing the *Constant Propagation* technique and from the documentation reading, it is assumed it is also able to initialize data before runtime.

6. **Other types of optimization.** This tool is great at *Dead Code Elimination*, which, as the name suggests, is a technique used to remove parts of the code that have no effect on the result of the program. Such cases include variables that are never used, or unreachable parts of the code.

   It also focuses on *Code Minification*, which is a technique that focuses on removing unnecessary spaces, line breaks, comments and other unnecessary characters in the code. This allows the program to be smaller, which saves time, increasing performance.

Chapter 3

A Performance Problem

The goal of this chapter is to provide the reader with a better understanding of the problem being tackled in this dissertation, along with the approach to be followed. The hypothesis to be proven and the assumptions made are described in this chapter as well. Finally, this chapter contains the methods to be used in the validation process of this dissertation.

3.1 Problem Description

As briefly mentioned in Chapter 1, the problem this dissertation is trying to tackle is the low performance verified on JavaScript programs.

The growth of the number of websites has been superior to that of users, which causes competition between websites for the user bases. Websites need users, usually to generate money. Even websites that are free to use can generate money just by showing ads on its pages. So, the important question on website owners' minds may be how to attract these users to a website, and the answer may very well be improving the performance of their websites

Low performance generates both higher computational costs and dissatisfaction on the user base. A bad user experience due to high loading times on a website can lead to abandonment since it is estimated that a staggering 53% of users abandon websites which take more than 3 seconds to load (Shellhammer).

3.2 Proposal

As this dissertation intends to tackle the low performance of JavaScript programs, the main goals are to study the Bentley Rules, their applicability and impact when applied to the JavaScript language, and to conjugate that study with a JavaScript code transformation, that both analyses the code and decides when to apply it.

In Chapter 1, the goals of this dissertation were mentioned. And, following that mention, in this section, they will be explained.

To solve this problem, the applicability of a set of rules called the Bentley Rules to the JavaScript language will be studied, and then, that application will be performed, with the development of a JavaScript code optimizer.

3.3 Hypothesis & Research Questions

This dissertation has one major hypothesis, which is divided into three different research questions. The hypothesis trying to be proven is: **the Bentley Rules are applicable to the JavaScript language.**

These rules were defined to be applicable to any program in any language, as they are very generalist in the types of rules they contain. However, the programming language JavaScript only first appeared in 1995 (Newswire, 1995), thirteen years after the Bentley Rules had been written. As a consequence, their applicability may not be possible.

The research questions this dissertation is trying to answer all depend on the hypothesis being true. The research questions are:

1. **Do the Bentley Rules, when applied to JavaScript code, increase the code's performance?** If the hypothesis is proven, then this is the first question to be answered. However, if it is found that the Bentley Rules are not applicable to JavaScript, then the answer to this question is automatically negative as well. This is because, if the rules cannot be applied to the JavaScript language, then it is not possible they are able to improve performance.

2. **Out of all the applicable Bentley Rules, which one has the best impact in the performance of JavaScript programs?** This question focuses on differentiating each Bentley Rule in terms of potential impact in the JavaScript code when applied to it. Therefore, it also depends on the hypothesis being proven.

3. **Is it possible to translate the Bentley Rules into automatic code transformations?** This question focuses on trying to apply the rules to JavaScript programs, by automatically analyzing it and modifying it to improve its performance. Like the previous questions, it also depends on the veracity of the hypothesis.

3.4 Assumptions

Throughout this dissertation, some assumptions will be made that are required to validate the results:

1. **The already developed tool used by Jscrambler in code analysis is perfect.** It will be assumed that the tool that transforms JavaScript code into *Abstract Syntax Trees* does not

fail or malfunction. The code written in the development of the optimization tool will not take into account these cases.

2. **The JavaScript programs that the optimization tool is going to be used on are well-written and can be fully analyzed by the existing code analyzing tool.** Although some time will be spent in reading the code, the focus of this dissertation is for the tool to perform optimizations and not to evaluate the accuracy of the code analyzing tool.

3. **The JavaScript programs that the optimization tool is going to be used on are well-behaved.** By this, it is meant that the code is not capable of reflection. Reflection is the program's ability to modify itself at runtime Vinoski (2005). Since the optimization will occur before runtime, these types of programs can be severely damaged if modified in the same manner as others.

4. **Although the JavaScript programs are well-written, their performance can always be increased.** If the program, after the tool modifies it, has not increased in performance, the conclusion taken from it will be that the tool is not working properly. The tool should never decrease a program's performance.

5. **The performance of a program is strictly related to its execution time.** When evaluating the performance gain or loss of each rule, the metric used to calculate this is the operations per second.

3.5 Validation

In Chapter 1, the goals of this dissertation were set. The first goal mentioned was the study of the applicability of each of the Bentley Rules to the JavaScript language, and the measurement of the respective impacts on said language. To measure their impact, test fixtures will be created. The validation of this goal is the creation of the test fixtures in itself, and their results, since if it is possible to create the test fixtures representing each rule, it means that the respective rule is applicable.

The second goal is the application of the results obtained in the path to achieving the first goal, which is the implementation of a code transformation relating to one of the rules which showed to generate a better improvement in the performance of JavaScript code. This will be done taking advantage of Jscrambler's product, which is capable of analyzing and transforming JavaScript code. To validate this part of the work, one open-source JavaScript project will be transformed with the newly created code transformation. Then, its performance will be compared with the performance of the same project, unaltered.

Chapter 4

Bentley Rules Study

This chapter has the purpose of describing the steps taken to achieve the goals previously mentioned in Chapter 1, as well as explaining why the steps taken were thought to be adequate to their specific situation. The first four sections are dedicated to the steps taken to achieve the first main goal, "Study the Bentley Rules and their applicability to the JavaScript language":

Section 4.1 contains the author's decision on the applicability of each Bentley Rule regarding their applicability to the JavaScript language, as well as what supports said decisions. Section 4.2 describes the process of creating the test fixtures for each rule that was deemed to be applicable.

The next section, Section 4.3, describes the development process of the application which was then used to evaluate the performance gain, or loss, of each rule, using the test fixtures described in the previous section. Section 4.4 describes the methodology behind the selection of the browsers in which the application would be tested in, as well as the steps taken for the testing of the application.

4.1 Study of the Bentley Rules

The author's take on each of the Bentley Rules is present in this section. The decision on the possibility of application of each rule to the JavaScript language is also present as well, accompanied by the reasons behind each decision.

The schema for this section follows the same schema as Section 2.3. At the end of the section, in Subsection 4.1.7 there is a table containing the decision for each rule, which functions as a summary of the entire section.

4.1.1 Space-for-Time Rules

Space-for-Time This subsection contains the decisions on the applicability of each Space-for-Time rule.

1. **Data Structure Augmentation** — This rule is applicable to the JavaScript language, as long as there is permission to access and modify the database. More than just the database structure would need to be modified, but the code related to the insertion of new database entries, as well as every entry already in the database tables that would be modified. If there is not the possibility to modify everything mentioned, then this rule is not applicable for that specific case.

2. **Store Precomputed Results** — The act of storing results during execution time for easier access in a future call of a function is more commonly known as memoization, and it is applicable to the JavaScript language. Similarly to the *Store Precomputed Results* rule, it only targets pure functions. It requires the creation of a data structure to store the results, as well as a verification, before the execution of a function, to decide whether it should be executed normally, or, in the case it had already executed for the same set of arguments, use the result stored in that execution.

3. **Caching** — This is not applicable to JavaScript, as the Cache object in this language is still an experimental technology as of the time of development of this dissertation (Mozilla, 2018).

4. **Lazy Evaluation** — This rule is applicable to the JavaScript language, however it goes against the idea behind the rules *Precompute Logical Functions* and *Compile Time Initialization*, whose intention is to precompute certain instructions to avoid doing so during the execution.

4.1.2 Time-for-Space Rules

This subsection contains the decisions on the applicability of each Time-for-Space rule.

1. **Packing** — Doing this may not decrease the space of the program structure, but instead decreases the space of the program storage. Despite being applicable to the JavaScript, like the *Data Structure Augmentation*, as long as there is access and permission to modify the data structures, and the code specific to the insertion and access to them, the effects of the application of this rule are not desired in the context of this dissertation, as it will increase the execution time of a program by performing unnecessary computations.

2. **Interpreters** — Much like the other Time-for-Space rules, the impact this rule has on a program is not desired in the context of this dissertation. Reducing the space of a program, and increasing the execution time goes against the definition of performance defined in Chapter 3.

4.1.3 Loop Rules

This subsection contains the decisions on the applicability of each Loop rule.

1. **Code Motion Out of Loops** — This rule is applicable to the JavaScript language, and it can be very beneficial performance-wise, but it can only be applied when the code has been poorly developed, as the code being moved out of a loop with *n* iterations would be, unnecessarily, executing *n - 1* times before the application.

2. **Combining Tests** — This rule is applicable to the JavaScript language, but its application is not easy to generalize, as the combination of the tests is always dependent on the tests themselves and it will vary from program to program.

3. **Loop Unrolling** — This rule is applicable to the JavaScript language, but its application should not be made to every loop. Loops that perform few iterations and with not very large bodies may be good candidates. However, loops with large bodies and several iterations should not be targeted, as the repetition of the large body many times may increase the size of the code to a point where it is no longer faster when compared to simply performing the unaltered loop.

4. **Transfer-Driven Loop Unrolling** — Despite applicable to the JavaScript language, the impact this rule is expected to have is nearly null, since the only thing it focuses on is the removal of assignments from the code, whose impact is not that great.

5. **Unconditional Branch Removal** — This technique is only applicable to lower-level languages since high-level languages have compilers capable of handling these cases effectively.

6. **Loop Fusion** — This rule is applicable to the JavaScript language. However, it can only be done when there is a very clear idea of how the loops and the code inside them work. In the example given, if accessing the age of a person would trigger an assignment to the next person's name, then those loops shouldn't be fused.

4.1.4 Logic Rules

This subsection contains the decisions on the applicability of each Logic rule.

1. **Exploit Algebraic Identities** — This rule is applicable to the JavaScript language, however its impact should not be very large. Detecting the cases where an expression can be simplified may not be straightforward, as well.

2. **Short-Circuit Monotone Functions** — This rule is applicable to the JavaScript language, and in the cases where long and costly functions are brought to a halt preemptively, it could have a very good impact on the performance of the program. However, in the cases where the function cannot be terminated before its last command, having evaluations on whether the function can be short-circuited, or not, throughout the function's body may hinder the performance.

3. **Reorder Tests** — This rule is applicable to the JavaScript language. For the cases where there are very few tests, the impact expected should not be very high. However, when there is a large sequence of tests being performed, placing the most probable at the top, thus preventing the others to take place, could be very beneficial to the code's performance.

4. **Precompute Logical Functions** — This rule is applicable to the JavaScript language, and can be seen as a more specific application of the *Store Precomputed Results*, as it is applicable only to functions. The functions this rule targets must be pure — function purity is explained in Section 2.6. Also, the domain of the function, as well as the type of the arguments received, must be known for the application to be possible.

5. **Control Variable Elimination** — Although this rule is applicable to the JavaScript language, it does not seem to have a very high potential in terms of performance gain as the only difference would be the removal of the assignment to the control variable.

4.1.5 Procedure Design Rules

This subsection contains the decisions on the applicability of each Procedure Design rule.

1. **Collapse Procedure Hierarchies** — This rule is applicable to the JavaScript language, and is commonly referred to as *Inline Expansion*. Although it reduces the number of instructions in a program, if a procedure is called several times during the execution of a program, inlining may not be the best approach, especially if it has a large body, as the program's size could increase drastically.

2. **Exploit Common Cases** — This rule is applicable to the JavaScript language. Predicting the common cases may not be possible, or there may not be common cases for some functions. However, when there are, this rule can have a very positive impact in the performance, as it removes a lot of computations from the execution.

3. **Use Coroutines** — This rule is applicable to the JavaScript language.

4. **Transform Recursive Procedures** — This rule is not applicable to the JavaScript language, as it focuses on modifying lower-level instructions.

5. **Use Parallelism** — This rule is not applicable to the JavaScript language, as it is a single thread language. To apply parallelism, a multi-thread language is required.

4.1.6 Expression Rules

This subsection contains the decisions on the applicability of each Expression rule.

1. **Compile-Time Initialization** — *Constant Propagation* is how this rule is more commonly known, and it is applicable to the JavaScript language. However, this is a technique that every JavaScript engine should be able to apply in the compilation of programs.

2. **Eliminate Common Subexpressions** — This rule is applicable to the JavaScript language. However, unless the expressions are used a large number of times, the impact of its application on the performance will be very small.

3. **Pairing Computation** — This rule is applicable to the JavaScript language. The possibility of application of this rule in a program usually indicates that the program has not been very well developed, since it should be a major concern of any developer to pair any computations that have high computational costs.

4. **Exploit Word Parallelism** — This rule is not applicable to the JavaScript language, as there is no way in this language to define the length of each word in the underlying architecture of the machine.

4.1.7 Summary of Decisions

In Table 4.1 is a summary of the decisions made in this section.

4.2 Test Fixtures Creation

After understanding which Bentley Rules were applicable to the JavaScript language, was the creation of test fixtures that accurately represent the before and after states of a part of a program when each rule is applied to it. The main goal of the test fixtures was to evaluate the potential change in performance that each rule has when applied to JavaScript.

The possible application scenarios of each rule is different, as some may have only one possible case of application, and others may have a very large number of possible applications. Therefore, to achieve more conclusive results, an effort was made to create more than one possible scenario for some of the rules. Hence, each rule has, at least, one test, which in turn contains two different test fixtures, one representing the state of the code before a rule is applied to it, which is the unoptimized code, and other representing the state of the code after the rule is applied, or the same code, but optimized.

As mentioned, some rules have more than one test, which is the case with all the *Loop Rules*, for example, as tests were made to test the impact of the rules in small loops and in large ones.

When the tests were executed, the part of the code that was considered to calculate the performance impact was the body of the functions. All the other parts of code outside the function bodies is neutral code, as it does not affect the calculations. Every test fixture that was created and later used in the testing can be found in the Appendix A.

4.3 Development of Performance Evaluating Application

The third part of the implementation was the development of an application, in NodeJS, to help with the evaluation of the performance gain or loss for each rule, resorting to the test fixtures

Name of the Rule	**Applicability**
Data Structure Augmentation	This rule is applicable to the JavaScript language.
Store Precomputed Results	This rule is applicable to the JavaScript language.
Caching	The application of this rule to the JavaScript language is impossible. This is due to the inability of this language of accessing the cache.
Lazy Evaluation	This rule is applicable to the JavaScript language.
Packing	This rule is not applicable as it is part of the Space-for-Time rules and they make programs slower.
Interpreters	This rule is not applicable as it is part of the Space-for-Time rules and they make programs slower.
Code Motion Out of Loops	This rule is applicable to the JavaScript language.
Combining Tests	This rule is applicable to the JavaScript language.
Loop Unrolling	This rule is applicable to the JavaScript language.
Transfer-Driven Loop Unrolling	This rule is applicable to the JavaScript language.
Unconditional Branch Removal	This rule is not applicable to the JavaScript language as it is only applicable to lower-level languages.
Loop Fusion	This rule is applicable to the JavaScript language.
Exploit Algebraic Identities	This rule is applicable to the JavaScript language.
Short-Circuit Monotone Functions	This rule is applicable to the JavaScript language.
Reorder Tests	This rule is applicable to the JavaScript language.
Precompute Logical Functions	This rule is applicable to the JavaScript language.
Control Variable Elimination	This rule is applicable to the JavaScript language.
Collapse Procedure Hierarchies	This rule is applicable to the JavaScript language.
Exploit Common Cases	This rule is applicable to the JavaScript language.
Use Coroutines	This rule is applicable to the JavaScript language.
Transform Recursive Procedures	This rule is not applicable to the JavaScript language as it is only applicable to lower-level languages.
Use Parallelism	This rule is not applicable to the JavaScript language.
Compile-Time Initialization	This rule is applicable to the JavaScript language.
Eliminate Common Subexpressions	This rule is applicable to the JavaScript language.
Pairing Computation	This rule is applicable to the JavaScript language.
Exploit Word Parallelism	This rule is not applicable, as JavaScript has no means to do this.

Table 4.1: Summary of verdicts on the applicability of each rule to the JavaScript language

created earlier, and to JSPerf, which is a JavaScript performance evaluator, thoroughly described in Chapter 2.

The application developed requires the test fixtures to be in a specific folder and have specific file names, and has several features. In ascending order of simplicity, the first of these features is enabling the user to execute a test for a specific test of a chosen rule. After the application uses JSPerf to evaluate the performance gain or loss, it generates a small report for that test. This report contains the following parameters:

1. **Ops/sec (Before & After).** These two values are calculated by JSPerf and, as their name indicates, it is the mean number of operations per second the test fixture achieved. This is calculated resorting to the time of execution of each of the respective test fixture's sample runs. These values are used ahead and are represented as Ops_B and Ops_A, respectively.

2. **Margin of Error (Before & After).** These two values are also calculated by JSPerf resorting to 95% confidence intervals. This means that there is a 95% chance that the real mean of Ops/sec of a test fixture is in the following interval: [Ops/sec - MOE, Ops/sec + MOE] These values are used ahead and are represented as MOE_B and MOE_A, respectively.

3. **Performance Impact (Worst, Best and Average Cases).** These three values are calculated by the application, which makes use of the preceding values to calculate them. Together, these three values represent a confidence interval for the performance impact of each rule. This resulting interval, since it combines the intervals from the test fixtures representative of Before and After, has a confidence of around 90%, which is the 95% squared. These values are calculated using the following formulas:

$$WorstCaseBefore = Ops_B(1-\frac{MOE_B}{100}) \tag{4.1}$$

$$BestCaseBefore = Ops_B(1+\frac{MOE_B}{100}) \tag{4.2}$$

$$WorstCaseAfter = Ops_A(1-\frac{MOE_A}{100}) \tag{4.3}$$

$$BestCaseAfter = Ops_A(1+\frac{MOE_A}{100}) \tag{4.4}$$

$$WorstCase = 100(\frac{WorstCaseAfter}{BestCaseBefore}-1) \tag{4.5}$$

$$BestCase = 100(\frac{BestCaseAfter}{WorstCaseBefore}-1) \tag{4.6}$$

$$AverageCase = 100(\frac{Ops_A}{Ops_B}-1) \tag{4.7}$$

The second of the features is the ability for the user to execute all the tests for a given rule. This generates reports for each test, and then a report for the rule, which uses the individual test reports. The report for the rule just contains three parameters, which are the averages of the best, worst and average cases of each test in that rule.

The third feature is giving the user the ability to execute all tests for all the rules consecutively, which generates all the reports mentioned before, as well as three different rankings for all the rules. These rankings are for the best, average and worst case averages included in each rule's report. In these rankings, the rules are ordered from the rule which generated the most gain in performance, to the rule which generated the least gain.

The final feature is giving the user the ability to check the reports generated for tests, rules or the rankings.

4.4 Browsers Selection & Test Automation

The selection of the browsers where the tests were to be run in was based on the usage of those browsers. Several sources were consulted to make a well-informed decision on this topic Statcounter (2018), Share (2018). According to these sources, the most used web browsers are Chrome, Safari, UC Browser, Firefox, Opera, Internet Explorer, Samsung Internet, and Edge.

Considering Chrome, Opera and Samsung Internet all use the same JavaScript engine, the v8, only the first was selected, since it was the most used. Another aspect to be taken into account is, since Microsoft has ended support for most versions of Internet Explorer Microsoft (2015), and is focusing more and more on the newer Edge browser, it was decided that the Internet Explorer browser was not to be tested, but the Edge browser was, regardless of its usage percentage being lower.

The UC Browser, at the time of this study did not allow for access to *localhost*, which made the testing impossible. Therefore, it was excluded from consideration. All things considered, the browsers selected for performing the tests were Chrome, Firefox, Edge and Safari.

After the selection of the browsers in which the application would run, it was time to do so. This was done by automatic *Selenium* tests, which were run resorting to a very helpful tool, *BrowserStack*, thoroughly described in Chapter 2, which supports test automation in several different browsers across several different devices and operating systems. The tests were developed with *Node.js*.

The tests were executed in just one machine, the author's personal computer. Before running the tests, a server needed to be started, and it contained all the test fixtures and the application to be run. This server was developed with the Express framework for Node.js. Then, in the same machine, the tests would be run.

The application, when run, would use the Browserstack platform to open the browser selected, access the local application previously created, and clicking the button which started the performance tests. These tests consisted of running every test fixture found in the directory of the application 100 times. When it was detected every test has been concluded, then the browser would be closed. This was done for each of the four selected browsers.

The author's machine, which was used to perform the testing, was an Asus GL552JX. It was running the Windows 10.1 OS, and had an Intel® Core™ i7 4720HQ processor, as well as 8GB RAM. The Performance Mode was active while the tests were running.

4.5 Results

This section contains all the results obtained after running the automated tests on each of the four selected browsers, Chrome, Firefox, Edge and Safari. Each test was executed 100 times, to make sure the results are statistically significant, and the margin of error is relative to a 95% confidence interval.

Tables 4.2, 4.3, 4.4 and 4.5 contain the results obtained in the Chrome, Firefox, Edge and Safari browsers, respectively. Then, in Subsection 4.6, the global results are presented and the author exposes his thoughts on these results, and their impact on the next part of this dissertation.

Table 4.6 shows the average of the worst, best and average cases for each rule in the four tested browsers. Each result was obtained by calculating the worst, best, and the average case for each rule in each of the browsers and calculating the average.

4.6 Conclusions

In this chapter, the hypothesis of this dissertation — "the Bentley Rules are applicable to the JavaScript language" — was supported. The applicability of the Bentley Rules to the JavaScript language was proven by the creation of the test fixtures. The possibility of recreating JavaScript code representative of the before and after states of the application of a rule demonstrates that said rule is applicable to the language.

Out of the 26 total rules, only 7 of them were considered inapplicable. Since these rules were defined to be very generalist, it is not surprising that the majority is applicable to the language.

The answers to the first and second research questions — respectively "Do the Bentley Rules, when applied to JavaScript code, increase the code's performance?" and "Out of all the applicable Bentley Rules, which one has the best impact in the performance of JavaScript programs?" — were found as well.

In Table 4.6, it can be observed that, in the worst case scenario, 6 of the 18 applied rules have a negative impact, while the other 12 have a positive one. In the average case, only 2 have a negative impact, while this is only verified in 1 when calculations are made for the best case scenario. With this data, it can be concluded that the impact of each rule always depend on the program, or part of the program that they're applied to. An example of this can be the Loop Unrolling rule, whose verified impact increased between loops with 25 iterations and 1000 iterations, when the body of the loop was a simple increment to a variable. However, between the same number of iterations, it was verified a decrease in impact when the body of the loop included accessing an array.

The rule which achieved the best impact, across all the browsers, in the performance of JavaScript code was **Precompute Logical Functions**, and it is due to the nature of the rule, which is avoiding many computations during the execution time, by performing the same computations in compilation time. Although it achieved outstanding results in Chrome, Firefox and Safari, in the Edge browser it achieved negative impact results.

Name of the Rule	Test Number	Before	MOE	After	MOE
Code Motion Out of Loops	1	125,696,361	3.32%	131,989,906	3.33%
Code Motion Out of Loops	2	296,054	3.02%	296,497	2.60%
Code Motion Out of Loops	3	3,014	2.61%	2,837	2.83%
Code Motion Out of Loops	4	1.52	2.82%	1.50	3.77%
Collapse Procedure Hierarchies	1	309,902	1.88%	285,873	2.74%
Collapse Procedure Hierarchies	2	30.53	2.31%	31.18	2.29%
Combining Tests	1	32,043,080	6.87%	36,707,254	2.33%
Combining Tests	2	5,188,075	2.24%	6,139,281	2.27%
Combining Tests	3	107,766	2.53%	145,320	2.47%
Compile Time Initialization	1	578,705,075	2.25%	575,141,451	2.50%
Control Variable Elimination	1	4,027,271	3.37%	6,319,409	2.63%
Data Structure Augmentation	1	1,406,673	2.49%	585,721,452	2.13%
Data Structure Augmentation	2	147,629,612	2.30%	575,407,159	2.85%
Eliminate Common Subexpressions	1	19,968,010	2.25%	32,361,442	2.52%
Eliminate Common Subexpressions	2	2,227,129	2.85%	6,515,513	2.83%
Exploit Algebraic Identities	1	41,782,328	3.09%	48,181,780	2.47%
Exploit Algebraic Identities	2	27,989,845	2.22%	27,964,205	2.19%
Exploit Common Cases	1	5,408,797	2.26%	8,288,166	2.30%
Lazy Evaluation	1	510,280,159	2.35%	479,822,945	2.58%
Lazy Evaluation	2	516,824,691	2.44%	502,551,590	2.01%
Lazy Evaluation	3	491,837,681	2.26%	492,034,119	2.14%
Loop Fusion	1	27,943,219	6.89%	32,360,123	4.70%
Loop Fusion	2	873,456	2.42%	1,628,160	2.30%
Loop Fusion	3	4,287	2.86%	5,888	2.51%
Loop Fusion	4	45.63	2.21%	56.04	1.90%
Loop Unrolling	1	26,994,911	3.69%	567,919,967	2.57%
Loop Unrolling	2	807,892	2.15%	143,520,045	3.01%
Loop Unrolling	3	2,303,995	2.49%	6,876,300	11.43%
Loop Unrolling	4	71,453	2.84%	2,325	2.86%
Pairing Computation	1	12,374,978	3.09%	15,735,046	2.74%
Precompute Logical Functions	1	129,000,145	2.72%	620,796,220	1.13%
Precompute Logical Functions	2	411,152	1.72%	569,000,420	2.46%
Reorder Tests	1	28,554,649	2.37%	30,054,661	2.37%
Reorder Tests	2	21,279,974	2.56%	21,124,386	2.27%
Short-Circuit Monotone Functions	1	93,995,809	4.35%	91,194,978	6.53%
Short-Circuit Monotone Functions	2	1,237,734	7.44%	1,295,604	6.84%
Short-Circuit Monotone Functions	3	14,973	3.59%	14,273	5.41%
Store Precomputed Results	1	69.10	2.46%	2,002	2.04%
Transfer-Driven Loop Unrolling	1	130,662,304	2.95%	133,872,749	1.84%
Transfer-Driven Loop Unrolling	2	827,475	2.16%	877,896	2.51%
Transfer-Driven Loop Unrolling	3	368,134	2.56%	406,771	2.93%

Table 4.2: Results obtained in the Chrome browser

Name of the Rule	Test Number	Before	MOE	After	MOE
Code Motion Out of Loops	1	10,162,014	6.01%	11,489,761	4.02%
Code Motion Out of Loops	2	109,271	2.66%	117,863	2.28%
Code Motion Out of Loops	3	842	2.56%	817	3.22%
Code Motion Out of Loops	4	8.79	2.17%	7.88	5.87%
Collapse Procedure Hierarchies	1	520,306	4.28%	537,656	2.39%
Collapse Procedure Hierarchies	2	56.87	2.37%	56.29	2.59%
Combining Tests	1	76,821,013	1.72%	68,033,114	1.54%
Combining Tests	2	4,854,205	1.37%	5,331,111	1.81%
Combining Tests	3	93,824	1.27%	109,811	1.93%
Compile Time Initialization	1	903,323,917	1.57%	847,413,233	2.23%
Control Variable Elimination	1	5,094,295	2.04%	5,249,417	2.15%
Data Structure Augmentation	1	568,436	1.47%	879,420,626	1.28%
Data Structure Augmentation	2	14,539,447	1.81%	880,020,894	1.27%
Eliminate Common Subexpressions	1	1,098,418	59.27%	30,656,228	3.58%
Eliminate Common Subexpressions	2	1,940,560	3.83%	4,253,820	4.18%
Exploit Algebraic Identities	1	853,676,645	1.11%	849,576,661	0.99%
Exploit Algebraic Identities	2	45,478,281	1.89%	40,237,141	1.81%
Exploit Common Cases	1	3,816,731	1.16%	6,373,330	1.14%
Lazy Evaluation	1	689,752,348	1.24%	697,953,815	1.04%
Lazy Evaluation	2	720,027,153	1.07%	696,782,160	1.20%
Lazy Evaluation	3	715,810,961	1.27%	716,954,162	1.12%
Loop Fusion	1	61,622,536	1.31%	96,538,598	2.73%
Loop Fusion	2	1,366,437	1.66%	2,291,314	1.67%
Loop Fusion	3	4,996	1.29%	5,923	3.28%
Loop Fusion	4	4.39	1.33%	4.50	1.01%
Loop Unrolling	1	22,737,412	1.43%	870,597,229	1.27%
Loop Unrolling	2	994,934	2.05%	79,040,913	1.15%
Loop Unrolling	3	1,591,801	14.48%	808,749,933	1.55%
Loop Unrolling	4	17,296	2.81%	6,758	3.03%
Pairing Computation	1	23,709,165	1.20%	32,122,333	1.16%
Precompute Logical Functions	1	241,134,138	1.04%	735,233,103	1.55%
Precompute Logical Functions	2	243,800	1.23%	730,325,108	1.78%
Reorder Tests	1	46,972,727	1.24%	50,334,302	1.74%
Reorder Tests	2	31,784,168	1.47%	31,420,998	1.08%
Short-Circuit Monotone Functions	1	89,604,650	3.30%	91,185,112	3.25%
Short-Circuit Monotone Functions	2	1,503,710	3.16%	1,511,820	3.98%
Short-Circuit Monotone Functions	3	15,112	3.55%	14,042	7.18%
Store Precomputed Results	1	33.49	0.41%	1,012	12.13%
Transfer-Driven Loop Unrolling	1	174,604,570	1.75%	146,114,700	1.62%
Transfer-Driven Loop Unrolling	2	345,657	0.80%	796,956	0.66%
Transfer-Driven Loop Unrolling	3	157,922	1.38%	369,222	0.98%

Table 4.3: Results obtained in the Firefox browser

Name of the Rule	Test Number	Before	MOE	After	MOE
Code Motion Out of Loops	1	3,758,524	6.62%	4,145,297	5.20%
Code Motion Out of Loops	2	47,229	5.32%	47,537	5.45%
Code Motion Out of Loops	3	394	3.56%	404	3.48%
Code Motion Out of Loops	4	2.56	2.59%	2.57	3.01%
Collapse Procedure Hierarchies	1	94,105	2.71%	93,044	2.65%
Collapse Procedure Hierarchies	2	9.72	1.81%	9.66	1.56%
Combining Tests	1	14,252,576	1.72%	13,997,040	1.85%
Combining Tests	2	5,829,983	1.43%	5,747,877	1.27%
Combining Tests	3	168,186	1.01%	168,386	1.15%
Compile Time Initialization	1	184,848,485	1.79%	185,641,528	1.71%
Control Variable Elimination	1	4,861,558	2.76%	6,537,230	2.30%
Data Structure Augmentation	1	1,473,967	3.03%	109,434,557	2.20%
Data Structure Augmentation	2	30,787,106	1.89%	112,438,518	1.27%
Eliminate Common Subexpressions	1	6,868,635	4.62%	7,797,483	5.65%
Eliminate Common Subexpressions	2	1,066,877	3.87%	1,357,913	4.62%
Exploit Algebraic Identities	1	9,231,635	1.58%	16,637,344	1.22%
Exploit Algebraic Identities	2	7,656,607	2.26%	7,826,759	2.67%
Exploit Common Cases	1	1,682,806	1.65%	2,462,874	1.27%
Lazy Evaluation	1	123,276,652	1.39%	77,096,419	1.56%
Lazy Evaluation	2	124,872,958	1.27%	62,713,721	1.04%
Lazy Evaluation	3	124,991,459	1.09%	62,256,201	1.06%
Loop Fusion	1	9,880,779	5.03%	10,949,437	4.33%
Loop Fusion	2	794,074	1.95%	1,390,869	2.28%
Loop Fusion	3	3,838	1.94%	6,297	1.60%
Loop Fusion	4	8.31	1.76%	8.32	1.46%
Loop Unrolling	1	32,994,750	0.85%	191,949,969	1.97%
Loop Unrolling	2	1,050,108	2.25%	182,407,355	1.23%
Loop Unrolling	3	559,904	3.35%	579,277	3.35%
Loop Unrolling	4	10,993	3.69%	3,195	3.73%
Pairing Computation	1	4,330,325	4.05%	4,700,689	3.46%
Precompute Logical Functions	1	83,033,312	1.06%	71,458,598	2.32%
Precompute Logical Functions	2	79,468	1.19%	73,270	1.24%
Reorder Tests	1	13,130,701	3.17%	13,568,624	1.54%
Reorder Tests	2	11,637,619	1.37%	11,609,489	1.43%
Short-Circuit Monotone Functions	1	35,240,372	1.79%	35,109,751	1.79%
Short-Circuit Monotone Functions	2	1,436,168	2.56%	1,615,754	1.96%
Short-Circuit Monotone Functions	3	13,223	3.23%	14,001	2.12%
Store Precomputed Results	1	14.79	1.58%	339	3.72%
Transfer-Driven Loop Unrolling	1	82,572,417	1.10%	74,847,832	0.92%
Transfer-Driven Loop Unrolling	2	161,141	1.17%	164,347	0.90%
Transfer-Driven Loop Unrolling	3	73,916	2.01%	77,997	2.12%

Table 4.4: Results obtained in the Edge browser

Name of the Rule	Test Number	Before	MOE	After	MOE
Code Motion Out of Loops	1	13,829,087	10.37%	15,865,364	6.85%
Code Motion Out of Loops	2	104,245	2.07%	101,206	1.68%
Code Motion Out of Loops	3	600	1.83%	590	1.59%
Code Motion Out of Loops	4	4.74	4.57%	4.42	4.30%
Collapse Procedure Hierarchies	1	44,357	1.57%	46,508	1.63%
Collapse Procedure Hierarchies	2	4.77	1.55%	4.80	1.29%
Combining Tests	1	3,690,433	3.99%	2,807,379	1.62%
Combining Tests	2	2,421,575	1.91%	2,725,508	2.04%
Combining Tests	3	98,116	2.04%	145,298	1.66%
Compile Time Initialization	1	71,587,096	3.33%	74,045,579	3.48%
Control Variable Elimination	1	6,182,508	1.73%	6,976,488	2.61%
Data Structure Augmentation	1	2,382,842	2.00%	63,765,448	3.75%
Data Structure Augmentation	2	34,253,400	4.27%	64,335,290	3.83%
Eliminate Common Subexpressions	1	8,398,570	7.14%	11,752,608	4.62%
Eliminate Common Subexpressions	2	969,922	2.92%	2,270,026	3.13%
Exploit Algebraic Identities	1	2,960,825	2.62%	4,418,621	2.42%
Exploit Algebraic Identities	2	2,155,635	2.62%	2,091,502	3.19%
Exploit Common Cases	1	2,572,488	2.72%	2,985,550	2.90%
Lazy Evaluation	1	66,965,305	5.45%	53,951,503	17.46%
Lazy Evaluation	2	66,501,213	5.94%	28,676,573	7.92%
Lazy Evaluation	3	65,965,533	5.31%	29,131,641	7.74%
Loop Fusion	1	18,465,952	2.77%	20,052,793	2.77%
Loop Fusion	2	1,098,505	1.60%	1,277,101	1.74%
Loop Fusion	3	5,725	1.07%	6,339	1.42%
Loop Fusion	4	7.75	1.69%	8.65	1.80%
Loop Unrolling	1	35,571,517	1.39%	64,837,950	19.16%
Loop Unrolling	2	2,051,344	1.46%	70,437,105	2.07%
Loop Unrolling	3	725,580	1.82%	695,016	2.89%
Loop Unrolling	4	7,261	2.01%	4,061	1.63%
Pairing Computation	1	12,644,734	11.29%	10,355,592	8.56%
Precompute Logical Functions	1	49,898,706	8.31%	63,282,949	4.49%
Precompute Logical Functions	2	399,787	4.32%	62,413,385	5.19%
Reorder Tests	1	4,065,615	2.90%	3,996,868	2.76%
Reorder Tests	2	3,843,740	2.85%	3,848,426	2.22%
Short-Circuit Monotone Functions	1	40,512,947	2.25%	38,295,402	2.67%
Short-Circuit Monotone Functions	2	1,928,322	2.22%	2,025,259	3.20%
Short-Circuit Monotone Functions	3	17,666	2.21%	19,509	2.74%
Store Precomputed Results	1	46.15	2.57%	415	1.53%
Transfer-Driven Loop Unrolling	1	59,376,994	5.95%	45,183,276	13.09%
Transfer-Driven Loop Unrolling	2	680,379	9.49%	1,151,423	1.54%
Transfer-Driven Loop Unrolling	3	129,500	1.61%	154,417	1.22%

Table 4.5: Results obtained in the Safari browser

Name of the Rule	Worst Case	Average	Best Case
Code Motion Out of Loops	-3.24%	1.43%	6.26%
Collapse Procedure Hierarchies	-2.20%	0.05%	2.31%
Combining Tests	7.57%	9.73%	11.90%
Compile-Time Initialization	-3.21%	-0.74%	1.74%
Control Variable Elimination	23.72%	26.82%	29.91%
Data Structure Augmentation	26,176.61%	26,580.44%	26,984.28%
Eliminate Common Subexpressions	88.42%	95.60%	102.79%
Exploit Algebraic Identities	14.08%	16.49%	18.90%
Exploit Common Cases	42.99%	45.66%	48.32%
Lazy Evaluation	-26.08%	-23.35%	-20.61%
Loop Fusion	28.44%	31.57%	34.71%
Loop Unrolling	6,321.94%	6,440.13%	6,558.32%
Pairing Computation	9.32%	13.27%	17.23%
Precompute Logical Functions	55,535.98%	56,731.72%	57,927.45%
Reorder Tests	-0.45%	1.51%	3.46%
Short-Circuit Monotone Functions	-2.29%	1.69%	5.67%
Store Precomputed Results	2,042.56%	2,177.59%	2,313.13%
Transfer-Driven Loop Unrolling	24.90%	27.48%	30.07%

Table 4.6: Average of the worst, average and best cases in all the browsers tested.

Considering it achieved similar results with the After test fixtures, when accessing the 3rd or the 1400th elements in the three first mentioned browsers, and there was a large gap in operations per second in the latter, it was likely that the negative impact of the rule was due to the Edge browser's engine inability to efficiently reading from very large arrays.

Besides this rule, three other rules generated an improvement of over one thousand percent (1000%), which means that they improve the operations per second by more than ten times. These other rules were **Data Structure Augmentation**, **Loop Unrolling**, and **Store Precomputed Results**.

There is a fifth rule which stands out, **Eliminate Common Subexpressions**, which achieved an improvement of nearly one hundred percent (100%), meaning that the operations per second were nearly doubled.

4.7 Limitations

In this chapter, the test fixtures created either came from ideas the author had at the time of studying each rule, or were examples given by Jon Bentley, together with the description of the rules. However, many other cases for each rule may be possible, and different test fixtures might have achieved different results.

The final results do not represent the true impact of each rule in a complete JavaScript program, instead representing the impact in the specific instructions' modification or replacement. To achieve a better understanding of each rule's impact on JavaScript programs, it would be required

to apply each rule to programs developed in this language, and these programs would need to represent the average JavaScript program. This, however, is not possible to predict, and as such, it is never possible to calculate the true impact of a rule and its application.

As mentioned in Section 4.4, the UC Browser does not allow *localhost* access. Therefore, the application developed to test the impact of each rule was not tested on this browser. Although very different results were not expected on this browser, it would have been interesting to test on more browsers.

Chapter 5

Implementation of Code Transformation

This chapter contains the steps taken into achieving the second goal of this dissertation, which intends to take advantage of the Jscrambler product's code analysis and transformation capabilities, to develop a code transformation based on the results obtained in the last chapter, and to validate its performance gain potential.

The first section contains the steps taken to choose the rule which would be translated to a code transformation, and the last section has the analysis of the impact that transformation has on some JavaScript projects.

5.1 Choice of the Transformation

Having achieved the first goal of this dissertation, it was then time to apply the newly acquired knowledge regarding which Bentley Rules produces the most gain in performance when applied to JavaScript code. As stated in Chapter 1, the second goal of this dissertation was to implement a transformation, taking advantage of Jscrambler's code obfuscation tool, which is capable of applying transformations to JavaScript code.

The choice of which Bentley Rule to be translated into a code transformation was based on the following parameters:

1. **Applicability to the JavaScript language.** The rules that were considered not to be applicable to the JavaScript language were not considered in this part of the dissertation.

   This parameter sees 6 of the 26 total rules automatically out of contention.

2. **Performance gain potential.** The rules that were proven to be more impactful in a positive manner to JavaScript code would have preference over the less impactful ones.

Looking at the results obtained in the last chapter, 5 rules stand out in terms of performance gain. These rules are, in descending order of the results observed, Precompute Logical Functions, Data Structure Augmentation, Loop Unrolling, Store Precomputed Results, and Eliminate Common Subexpressions.

3. **Generalization Possibility.** Generalization is essential when implementing code transformations as there is a need to catch every case in which the transformation is to be applied. Also, if a rule is not generalizable – is only applicable in some specific cases – perhaps it will not really have a great impact.

   To generalize **Precompute Logical Functions**, every time a pure function is detected and its domain is known, the results for the function should be calculated, and the function call should be replaced by a look-up on the data structure used to store the results. The main possible complications with the generalization of this rule are determining the purity of a function and its domain.

   The generalization of **Data Structure Augmentation** is done by detecting database accesses and detecting how the data received is handled. Whenever it is detected that the data received is used to perform further calculations, then the database should be augmented to include these new values. However, this may not even be possible to implement, since there may not be permission to change the database or even change the parts of the code which add new entries to the database. Therefore, this rule is excluded from consideration.

   From the 5 rules mentioned, **Loop Unrolling** is the easiest to generalize, as it is simply needed to take the code inside a loop and repeat it for the number of iterations the loop is supposed to perform. However, in some cases, it might be difficult to predict the loop's behavior since it may jump some iterations, either backward or forward, and it is also hard to predict when the overhead generated by repeating the lines of code stop being beneficial.

   The generalization of the **Store Precomputed Results** rule is done by transforming pure functions. Instead of their normal behavior, they first verify if the function has already been called with the same set of arguments before. If it has, then returns the result, which had been stored, otherwise it executes normally, storing the result in the end. Like the Precompute Logical Functions rule, there is also the need to determine the purity of a function.

   Finally, the generalization of the **Eliminate Common Subexpressions** rule is achieved by detecting repeated evaluations of the same expression throughout the code. When its value is not altered, then the result of the first evaluation is stored, and used instead of the following evaluations. However, in JavaScript is impossible to predict, with static analysis, whether the result of an evaluation will be changed. Therefore, and since this rule has the lowest performance gain of all the 5 being considered, it is excluded from consideration.

4. **Usefulness in Jscrambler's context.** Considering Jscrambler has a product being sold to several companies throughout the world, and that this product is being used to help with the

implementation, it would make sense to implement a transformation that could help improve the product further.

From the 3 rules still in contention, the Jscrambler product is already capable of performing Loop Unrolling, which excludes it from consideration. Therefore, remain two rules, Precompute Logical Functions and Store Precomputed Results, and the difference between them is that the first requires knowledge on the domain of the function. Since this may rarely be possible, the picked rule was Store Precomputed Results, as it is most likely applicable to more functions than the Precompute Logical Functions rule.

5.2 Memoization Transformation

As mentioned in the last chapter, Store Precomputed Results is more commonly known as memoization, and as such, throughout the remainder of this document, this rule will be described as memoization.

The first step in implementing a code transformation was defining what parts of JavaScript code the transformation should target. In the case of memoization, it should only be applied to pure functions. Therefore, this generates the consequent question of whether all pure functions or just a specific set of pure functions should be targeted.

5.2.1 Target Definition

Summing up, memoization stores every computed result from a function during the execution, to access those results in a future occasion when the same set of arguments is used, which renders memoization useless whenever a function is called only once throughout the execution of the program or is called once for each set of arguments.

An example of this would be a function that receives a number, returns its square, but the program uses it to calculate the square of every number from 1 to 100. In this case, the function will never be called twice for the same arguments, not only making memoization useless but having the opposite effect to what was pretended. The additional verification of whether there is already a stored result for the arguments received will make the program increasingly slower, the more stored results there are.

Another concern regarding the application of memoization is the overhead caused by storing too many entries in the data structure specific for that purpose, consequently making the searches on the data structure more costly.

Considering the examples given, it is easy to understand that memoization will not have a positive effect in every pure function when applied, and should not be applied to any function that is considered pure.

However, despite the Jscrambler tool's capabilities of recognizing pure functions and transform them, it is still not able to predict when one of the concerning cases is going to happen. This is due to the nature of static analysis, which does not allow data and control flow analysis.

Data Structure	Num. of Samples	Average Ops/sec	Margin of Error
Array	100	1.246.218	0,99%
Map	100	1.414.216	0,84%
Object	100	1.223.003	1,52%

Table 5.1: Data Structure results

Therefore, it was decided that without user interaction, the tool would apply the transformation to every pure function that was detected. The users still have the ability to perform annotations, and disable a specific transformation for the entirety of the program, or for specific functions.

5.2.2 Data Structure Selection and Prototype

The first idea for the transformation was the creation of a memoization wrapper function, which can then be used by every function to be memoized. The wrapper function would create a data structure in which the results would be stored for the arguments received, and the return statement of the wrapper function would be the verification of the existence of a result for the arguments received, return it in case it existed, or executing the function normally in case it did not.

That raised the question as to which data structure would be more suitable for this case, in which there is a pair of key and value needing to be stored, and the key to be searched.

Once again JSPerf was used, this time to compare the performance of different data structures in JavaScript, by performing insertions and searches on them. The data structures considered were Arrays, Maps and Objects, and the test fixtures used to evaluate them are shown in Appendix B.

Table 5.1 shows the results obtained on the tests made to the different data structures, and it can be concluded that having a Map as the structure to store the results of the functions is the most performing of the options.

After selecting the Map data structure, which was proven to be the best choice, there was the possibility of designing a prototype of the memoization function, which would be used. Note that in this function, the key is a serialization of the arguments received.

```
var memoization = function (func) {
    var map = new Map();
    return function () {
        var key = JSON.stringify(arguments);
        var value = map.get(key);
        if (value == undefined) {
            value = func.apply(null, arguments);
            map.set(key, value);
        }
        return value;
    };
};
```

Listing 5.1: Prototype of the memoization function

5.2.3 Transformation Development

Having defined the target for our transformation, as well as how the code is supposed to look like after a transformation has occurred, it was then time to use the Jscrambler tool to develop the transformation. As the tool transforms the code into Abstract Syntax Trees, and the transformations are accomplished by modifying those AST.

The first goal of the transformation is detecting the pure functions, and to do so, the following steps were followed:

1. **Find the function declarations.** Function declarations in the Abstract Syntax are nodes called "FunctionDeclaration", and for this transformation, they are the only nodes the Jscrambler tool visits, ignoring all other types of nodes.

2. **Create auxiliary variables.** Some variables had to be created to help with the detection of impure functions. Initially, an array to store every function declaration that was found, then, inside the node of each function, arrays to store all variables that were either initialized inside the function's body or received as an argument and every function call inside the body. Lastly, a variable which would work as a flag and that stored the decision on the eligibility of the function. Whenever a red flag was found inside the function's body, it would mark this variable as true, and it would not be targeted.

3. **Verifying number of arguments, and return statements.** For each function, the first thing the tool does besides the creation of the auxiliary variables, is the verification on the number of arguments received by the function, which was achieved by simply evaluating the expression: "node.params.length > 0", and the verification of the presence of at least one "ReturnStatement" node inside the body of the function. If one of these two conditions is not verified, the function is considered ineligible.

4. **Verifying the function is pure.** To do this, the initialized variables will be of use. To detect when the function modifies or accesses variables outside its scope, whenever a node with a variable that is not present in the array for the acceptable variables, the function is immediately considered impure. The same goes for functions called inside its body. If they are not present in the array where the function declarations were stored, then the caller is instantly considered impure. In the case they are included in the array, then the called function's node is verified for its purity. If it's impure, so is the caller function.

5. **Inserting the memoization function node.** After every function has been verified for their purity, in the case there's at least one function considered eligible, or pure, then a node is inserted in the tree, containing the memoization wrapper function, as shown in the excerpt in the last subsection.

6. **Modifying the eligible functions' declarations.** Every function that is considered eligible is then modified so that it calls the memoization function, passing the function as argument. The following JavaScript code excerpt shows a pure function declaration, before and after the transformation:

```
1   function beforeMemoization(n) {
2       if (n === 0 || n === 1)
3           return n;
4       else
5           return fibonacci(n - 1) + fibonacci(n - 2);
6   }
7
8   var afterMemoization = memoization(function(n) {
9       if (n === 0 || n === 1)
10          return n;
11      else
12          return fibonacci(n - 1) + fibonacci(n - 2);
13  });
```

Listing 5.2: Prototype of the memoization function

As mentioned in the previous steps, whenever a red-flag was detected, the functions were always considered to be impure. This pessimistic approach had to be followed, due to the impossibility of performing dynamic analysis. With static analysis alone it is not possible to predict the impact a single instruction will have later in the code.

5.2.4 Compliance Testing

The Jscrambler product has the ability to test the transformations developed with compliance tests, which apply the desired transformations to several libraries and then test them to verify that their behavior remains unaltered. The libraries the transformation was applied to are: Esprima, Lebab, Acorn, Moment, jQuery, Knockout, Express, SVG.js, Chart.js and jsdom. During the compliance

tests, it was discovered that the memoization function idealized would need to suffer some changes to cover some specific cases:

1. **Circular references in objects.** Circular references happen when an object contains a reference to itself. Usually these cases were due to the fact that some objects in the test libraries were trees, whose nodes had references to the parent nodes, which in their turn had references to the child nodes. This brought the need of creating a function to detect object type variables and serialize them. To prevent circular references, the newly created function searches through the objects, storing every property in an array, and when a repeated property is detected, then it is ignored, not being serialized.

2. **Serialization of maps.** It is not possible to serialize maps using *JSON.stringify(map)*, where *map* is the name of the variable, in the same way as other objects or even primitives, instead being necessary to create a verification for these cases, and handle them accordingly, using *JSON.stringify([...map])*.

3. **Serialization of objects with non-enumerable properties.** Serializing was perhaps the major obstacle on the development of the transformation. The serialization is not possible by simply using *JSON.stringify(map)*. The transformation does not handle these cases, which is why the transformation requires annotations from the user to prevent functions which use objects with non-enumerable props that are required in the serialization from being transformed.

Code Listing 5.3 shows the memoization function that resulted in considering the mentioned cases, and it covers the first two, as the third was not implemented.

Seeing that the three mentioned cases together were not detected very often, and that in the majority of the cases where the memoization function was applied, it worked without any problem, it was decided that two different memoization function were to be used. The default, the function shown in Code Listing 5.3, was to be applied to every pure function detected, in case no annotation was found or other functions which had been enabled for memoization. The memoization function shown in Code Listing 5.1 was to be applied to function which was enabled for it, with annotations.

```
var memoization = function (func) {
    var map = new Map();

    function generateKey(args) {
        let str = ''; let i = 0;
        [].forEach.call(args, function (el) {
            str += '[' + i + ']:'; i++;
            if(typeof el === 'object'){
                if(el instanceof Map){
                    str += JSON.stringify([...el]);
                } else {
                    let cache = [];
                    str += JSON.stringify(el, function(key, value) {
                        if (typeof value === 'object' && value !== null) {
                            if (cache.indexOf(value) !== -1) return;
                            cache.push(value); }
                        return value; });}
            } else str += JSON.stringify(el);
        }); return str;
    }

    return function () {
        var key = generateKey(arguments);
        var value = map.get(key);
        if (value == undefined) {
            value = func.apply(null, arguments);
            map.set(key, value);
        }
        return value;
    };
};
```

Listing 5.3: Memoization function after compliance testing

5.3 Testing the Transformation

This section focuses on describing the steps taken to test the impact of the transformation developed, and it contains three subsections.

Subsection 5.3.1 describes the process of selecting the project to which the transformation was applied. Subsection 5.3.2 describes the process of applying the transformation to the selected project, and Subsection 5.3.3 describes the process of measuring the impact of the transformation.

5.3.1 Project & Browser Selection

The project selected was a car racing, open source game. Besides this being a well-known type game, the reasoning behind this choice was that games usually have higher requirements in terms of performance, and the one selected is part of a repository of a selection of different HTML5/JavaScript games (Gordon).

To test the application, the Chrome browser was used. This was due to this browser being the most used browser, as mentioned in the last chapter, and the nature of the game, which does not allow for automatic testing.

5.3.2 Setup

After cloning the repository into the machine where the tests would occur — the same machine described in Chapter 4 — the game was running at 60 frames per second (fps). To verify some improvement, it was necessary to cause the frame rate to be lower, otherwise, in the Chrome browser, any optimization would not be noticed.

To lower the frame rate, the number of cars was increased to 4000, the resolution was increased to 1920x1080, and the number of lanes in the road was also increased to 10. Then, in-game, it was possible to increase some sliders, which would decrease the performance of the game.

The next step was to apply memoization to the game. When the obfuscation and optimization tool was applied to the program, only enabling memoization, it targeted several different functions in the project. However, when the game was launched, the frame rate was 0 fps. This was due to some of the functions being constantly called, for every car in the game, which at this time were 4000, for every frame. Storing so much data caused the application to have this terrible performance.

After analysing the code, and realizing this, it was possible to disable memoization to some of the functions which were causing this. Then, the frame rate went up again. To measure the frame rate, a **console.log()** instruction was added to the function that was calculating it every second. A different **console.log()** instruction was added whenever the car crossed the finish line, to differentiate the frame rate in each lap.

5.3.3 Performing the tests

The tests were executed by the author, by playing the game. To prevent differences in style of play between executions, the approach followed was to always go in a straight line, unless when cars were in front. In these cases, a lane switch would have to occur.

As the game does not end, since it does not have a concrete goal, besides trying to beat a previous lap-time record, two laps were completed per execution. The two laps serve to notice if there are differences from one lap to another, as one of the functions which was modified stored values in the first lap, that were then used on the ensuing ones.

Run Number	Lap Number	FPS Average
1	1	42,24
1	2	42,94
2	1	44,65
2	2	45,25
3	1	45,18
3	2	46,48
4	1	42,91
4	2	43,31
5	1	46,13
5	2	46,61
6	1	45,91
6	2	46,81
7	1	42,59
7	2	43,80
8	1	46,97
8	2	46,54
9	1	46,44
9	2	45,26
10	1	46,80
10	2	45,53

Table 5.2: Results obtained in the runs without transformation

The game was executed 20 times, 10 of which without the transformation applied, and the other 10 with memoization applied. The executions were intercalated as to avoid one of the test cases to benefit from a better performance from the machine.

5.4 Results

In Table 5.2, it is possible to see the average frames per second obtained in each run, and each lap performed with the game without the transformation applied. Table 5.3 shows the same results but for the runs made with the game which had the transformation applied.

Then, in Table 5.4, it is possible to observe the combined results. These results are, in order seen in the table, the average fps obtained in all the first laps performed with the game without transformation, then the same for the second lap. The third row shows the average for all the measured fps in the 10 runs without transformation. This includes all the laps performed. The next three rows of the table show these results but with the transformation applied to the game.

5.5 Conclusions

In Table 5.2, it is possible to observe that the average fps count increased between the first and second laps of almost every run. The same can be observed in Table 5.3. This increase, when

Run Number	Lap Number	FPS Average
1	1	46,66
1	2	49,34
2	1	47,81
2	2	48,66
3	1	47,28
3	2	49,04
4	1	47,78
4	2	48,31
5	1	47,53
5	2	48,41
6	1	48,73
6	2	48,45
7	1	48,34
7	2	48,38
8	1	47,72
8	2	48,10
9	1	47,88
9	2	48,38
10	1	47,70
10	2	48,63

Table 5.3: Results obtained in the runs with transformation

Transformation	Lap	FPS Average
No	1st	44,67
No	2nd	45,06
No	1st & 2nd	44,86
Yes	1st	47,67
Yes	2nd	48,57
Yes	1st & 2nd	48,11

Table 5.4: Results obtained by combination

combining all the first laps, and the second laps is only equivalent to a one percent increase in the game without the transformation. However, in the transformed game, the verified increase is nearly two percent.

This increase is believed to be because there is a function which was transformed, which is responsible for the formatting of the time of each lap. During the first lap of each run, that function is storing every result, as the same value is not repeated during that lap. But in the second run, it will use the results stored, until the time of the first lap is exceeded. Although the computations required to format the time of a lap are not extensive, avoiding them can cause this slight increase.

The most interesting data that can be gathered from the results tables is the increase in average fps between the non-transformed and the transformed games. While the first only achieved 44,86, the second was able to achieve an average of 48,11. This equals a 7,24% increase in average fps.

The increase verified between the non-transformed game's first laps and the transformed game's first laps is around 6,72%, while the value observed for the second laps is 7,79%. The difference in the increases can be the cause of the already mentioned function.

Therefore, it can be concluded that this transformation, when applied to some functions, can have a very positive impact in the performance of JavaScript programs.

5.6 Limitations

With the compliance testing some cases where functions were wrongly considered pure were detected, and promptly corrected. However, it is believed there may be some very specific cases where the tool will consider impure functions as pure. The other side of the coin is verified as well, as a pessimistic approach was followed. Whenever something was detected in the code that had the potential to make the function impure, the function was always considered impure. With these limitations, it is believed dynamic analysis is required to detect function purity more accurately.

The transformation was applied and tested with only one project, as this is very time consuming, without the ability to perform automatic testing. With different projects, results may have been very different, for better or worse.

Although this transformation achieved positive results, this was only achieved after the author explored the program's code, as some of the functions that were transformed made the game unplayable. This was due to the overhead caused by storing so many values. With profiling, this could be avoided and the transformation of these functions could be avoided.

Chapter 6

Conclusions and Future Work

This chapter is a reflection upon the work done throughout the development of the dissertation. In the first section, the main difficulties felt with the development are described, followed by the main contributions of this work, the conclusions drawn, as well as ideas for possible future work in the last section.

6.1 Main Difficulties

The major challenge of this dissertation was in the creation of the test fixtures aimed at verifying the impact each of the rules has on JavaScript programs. This challenge comes from the fact that no program is equal to another, and the verified impact of a rule will always depend on the program itself, on the way the rule is applied to it, and on the engine the program will be running on, as well.

The detection of a function's purity in a language such as JavaScript was undoubtedly one of the hardest tasks of this dissertation. The dynamic nature of this language makes it nearly impossible to detect pure functions with static analysis. By running the compliance tests, many cases which made functions impure were detected, and changes were made to the code which applies the transformation. However, it is still believed that not all cases of impure functions are detected. Another effect of the same difficulty was the pessimist approach followed in the detection of pure functions. Therefore, some pure functions may not be detected as well.

The selection of the open-source project to apply the transformation to was also a difficult task. Although there are many open-source projects, a popular game project was preferred. This was due to the fact that performance is even more important in games, compared to websites. However, many of the games found either required really intricate playing skills, or did not contain any function that was targeted by the transformation.

6.2 Main Contributions

The main contributions of this dissertation to the field of Software Engineering are the following:

1. **A study about the applicability of the Bentley Rules to JavaScript.** Although some of the Bentley Rules are commonly applied to JavaScript, as is the case with Store Precomputed Results, or Loop Unrolling, among others, there has not been a study focused in all the rules' application.

2. **An application for performance evaluation across different browsers.** This application was used to test the impact of the Bentley Rules in four different browsers. By creating different test fixtures, it is possible to do the same for any possible code transformations.

3. **A code transformation.** By using the Jscrambler tool, the code transformation implemented has the potential to be applied to, and increase the performance of a wide variety of websites and applications.

6.3 Conclusions

The main conclusion drawn after the development of this dissertation is that the Bentley Rules, despite defined before its creation, are, in their majority, applicable to the JavaScript language. The hypothesis of this dissertation, described in Chapter 3, is therefore supported.

Moreover, it was shown that even popular, open-source JavaScript projects can be improved performance-wise. Spending time considering improvements to a program's code can have a very positive impact in the user experience, which can lead to having a more loyal user base.

With the tests made with the test fixtures for the Bentley Rules, described in Chapter 4, it was shown that the answer to the first research question is positive — the Bentley Rules, when applied to the JavaScript language, can have a positive impact in the performance. The Bentley Rule which showed the biggest impact was **Precompute Logical Functions**, thus being the answer to the second research question.

The answer to the third research question is also positive, as a code transformation was implemented, which improves JavaScript programs' performance. Although, without further improvement, it can target certain functions which will cause the program to run much more slowly. With a good knowledge of the code, however, it is possible to apply it to the most convenient functions, which will generate more gains performance-wise.

Both goals set in Chapter 1 were achieved, however some improvements could be made, which is the next section's focus.

6.4 Future Work

The following ideas for future work are believed to be, by the author, the next steps to be taken for the improvement of this project:

1. **Improving the methodology.** The methodology followed for this dissertation, with the creation of the test fixtures can be improved. Instead, modifying existing JavaScript project

in accordance with each rule would be a more reliable way to measure the impact of each of the rules.

2. **Testing for the combination of rules.** Some rules did not show to have a great impact on JavaScript code's performance. However, combining two rules could perhaps have a greater impact than the sum of the two impacts combined.

3. **Improve the understanding of analyzed code for increased customization of the memoization functions.** Profiling would allow a better understanding of how a certain program behaves, and, consequently, annotations would not be needed. Furthermore, having specific memoization functions for some functions, as is the case with functions which use non-enumerable props from an object, would mean a better performance gain. Storing simply the props of the object that make it different from others, would save some storage space, and make the reading and writing faster.

4. **Study the relation between overhead and time saved.** Although memoization has the potential to improve the speed of the execution of JavaScript code, the memory it consumes may be too large. Studying this trade-off between speed and memory should generate good knowledge on when to apply memoization.

5. **Implementing more code transformations.** Code transformations help developers improve their code's performance without having to spend much time worrying about it. A code transformation was implemented in the development of this dissertation, but there were some rules which showed a greater impact in JavaScript code, and translating them to code transformations is an idea for future work.

6. **Combine the code transformation with obfuscation.** Applying the code transformation along with code obfuscation could provide better insight to the code transformation's real impact in Jscrambler's context, as the optimization is seen as a measure to counter the negative impact of obfuscation in the programs' performance.

7. **Test the impact of the code transformation combined with other optimization tools.** Seeing the difference in impact of the code transformation when applied before, and after other JavaScript optimization tools have done their work, could prove to be even more beneficial for the performance of JavaScript programs. Tools as the ones mentioned in Chapter 2 could be a good starting point.

8. **Account for energy consumption.** As it was mentioned in Chapter 3, in this dissertation it was assumed that the performance of a program was only dependent of its execution time. However, taking into account both memory and energy consumption would perhaps lead to different results.

Appendix A

Bentley Rules Test Fixtures

This appendix contains the test fixtures for the Bentley Rules. The order in which they are presented follows the order they were mentioned and described in chapter 2. Not all of the Bentley Rules' test fixtures that were tested are present in this appendix, as is the case of the rule *Loop Unrolling*, instead containing a brief explanation of the test fixtures.

Each section in this appendix contains up to three subsections, named "Setup", "Before" and "After". The "Setup" subsection represents the parts of the test fixtures that are required for them to be executed, and do not have any influence in the results obtained, as the execution of these parts is not considered in the calculation of the time. The "Before" and "After" subsections represent, respectively, and as the names suggest, parts of code before and after a rule is applied to it, and are the parts that are executed and used to calculate the gain or loss in performance for the respective rule.

In the cases where only one test was created for each rule, or more than one test where the function called is the same, the name of said function is *before()* and *after()*. When there is more than one example and each one has a different function, the function names are the same as in the first case, but followed by the ordinal number correspondent to the test, for example, *beforeFirst()*, *afterSecond()*.

A.1 Data Structure Augmentation

A.1.1 Setup

```
function Person1 (info) {
    this.name = info[0];
    this.birthDate = info[1];
}

function Person2 (info) {
    this.name = info[0];
    this.birthDate = info[1];
    this.age = 0;
    this.birthDay = this.birthDate.getDate();
    let today = new Date();
    if(today.getFullYear() > this.birthDate.getFullYear()) {
        if(today.getMonth() > this.birthDate.getMonth() || (today.getMonth() ===
            this.birthDate.getMonth() && today.getDate() >= this.birthDate.getDate
            ())) {
            this.age = today.getFullYear() - this.birthDate.getFullYear();
        } else this.age = today.getFullYear() - this.birthDate.getFullYear() - 1;
    } else this.age = 0;
}

let name = 'Filipe';
let birth = new Date(Date.UTC(2017,8,12,6));
var person = new Person1([name, birth]);
var person2 = new Person1([name, birth]);
```

A.1.2 Before

```
function beforeFirst() {
    let today = new Date();
    if(today.getFullYear() > person.birthDate.getFullYear()) {
        if(today.getMonth() > person.birthDate.getMonth() || (today.getMonth() ===
            person.birthDate.getMonth() && today.getDate() >= person.birthDate.
            getDate())) {
            return(today.getFullYear() - person.birthDate.getFullYear());
        } else return(today.getFullYear() - person.birthDate.getFullYear() - 1);
    } else return(0);
}

function beforeSecond() {
    return person.birthDate.getDate();
}
```

A.1.3 After

```
function afterFirst() {
    return person2.age;
}

function afterSecond() {
    return person2.birthDay;
}
```

A.2 Store Precomputed Results

A.2.1 Setup

```
var max_fibonacci = 1476;
var fibonacciArray = [];

function fibonacci(num) {
    var a;
    var b;
    var c;
    if(num === 0 || num > max_fibonacci)
        return 0;
    else if(num <= 2)
        return 1;
    else{
        a = 1;
        b = 1;
        for(var i = 3; i <= num; i++){
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

A.2.2 Before

```
function before() {
    var results = [];
    for(let i = 0; i < 10000; i++) {
        let num = Math.floor(Math.random() * max_fibonacci) + 1;
        results.push(fibonacci(num));
    }

    return results;
}
```

A.2.3 After

```
function after() {
    var results = [];
    for(let i = 0; i < 10000; i++) {
        let num = Math.floor(Math.random() * max_fibonacci) + 1;
        if(fibonacciArray[num] == null) {
            fibonacciArray[num] = fibonacci(num);
        }
        results.push(fibonacciArray[num]);
    }

    return results;
}
```

A.3 Lazy Evaluation

A.3.1 Before

```
function before(rate) {
    var sum = 1 + 1;
    for (let i = 0; i < 10; i++) {
      if(i >= rate)
        return 3;
      else return sum;
    }
}
```

A.3.2 After

```
function after(rate) {
    var sum = () => 1 + 1;
    for (let i = 0; i < 10; i++) {
      if(i >= rate)
        return 3;
      else return sum();
    }
}
```

A.4 Code Motion Out of Loops

A.4.1 Setup

```
var final = "";
```

A.4.2 Before

```
function before(num_iterations) {
    for (var i = 0; i < num_iterations; i++) {
        var str1 = "Iteration";
        var str2 = " Number = ";
        var str4 = str1 + str2;
        final += str4 + i + "\n";
    }
}
```

A.4.3 After

```
function after(num_iterations) {
    var str1 = "Iteration";
    var str2 = " Number = ";
    var str4 = str1 + str2;

    for (var i = 0; i < num_iterations; i++) {
        final += str4 + i + "\n";
    }
}
```

A.5 Combining Tests

A.5.1 Setup

```
var numbersList = [];

for(var i = 0; i < 10000; i++) {
    numbersList.push(Math.floor((Math.random() * 10000) + 1));
}
```

A.5.2 Before

```
function before(numElements) {
    var i = 0;
    var numToSearch = Math.floor(Math.random() * 10000) + 1;

    while(i < numElements && numbersList[i] != numToSearch) {
        i++;
    }
    return i;
}
```

A.5.3 After

```
function after(numElements) {
    var i = 0;
    var numToSearch = Math.floor(Math.random() * numElements) + 1;

    numbersList[numElements] = numToSearch;

    while(numbersList[i] != numToSearch) {
        i++;
    }
    return i;
}
```

A.6 Loop Unrolling

This section does not contain the test fixtures for the *Loop Unrolling* rule, instead containing a brief explanation of what the test fixtures consisted of, as the test fixtures themselves would take an astounding amount of space in the document. In the "after" cases of this rule, instead of having a loop with n iterations, the code inside the loop was just repeated n times.

In this rule's test fixtures, four different cases were considered. Two of them are loops where a variable is increased by 1 each time the loop in executed. The number of iterations varied in each case. The other two cases consisted of loops which would iterate over an array and calculate the sum of its elements, each with a different number of iterations.

A.7 Transfer-Driven Loop Unrolling

A.7.1 Setup

```
var max_fibonacci = 1476;
```

A.7.2 Before

```
function before(num) {
    var a;
    var b;
    var c;
    if(num == 0 || num > max_fibonacci)
    return 0;
    else if(num <= 2) return 1;
    else{
        a = 1; b = 1;
        for(var i = 3; i <= num; i++){
            c = a + b;
            a = b; b = c;
        }
        return c;
    }
}
```

A.7.3 After

```
function after(num) {
    var a;
    var b;
    if(num == 0 || num > max_fibonacci)
        return 0;
    else if (num <= 2)
        return 1;
    else {
        a = 1; b = 1;
        for (var i = 1; i < num / 2; i++) {
            a = a + b;
            b = b + a;
        }
        if(num % 2) return a;
        return b;
    }
}
```

A.8 Loop Fusion

A.8.1 Setup

```
var numbersList = [];

for (var i = 0; i < 10000; i++){
    numbersList.push(Math.floor(Math.random() * 10000) + 1);
}
```

A.8.2 Before

```
function before(numElements) {
    var nums = [0 , 0];

    for (i = 0; i < numElements; i++) {
        if(numbersList[i] % 2)
            nums[0]++;
    }

    for (i = 0; i < numElements; i++) {
        if(numbersList[i] > (numElements / 2))
            nums[1]++;
    }
}
```

A.8.3 After

```
function after(numElements) {
    var nums = [0 , 0];

    for (i = 0; i < numElements; i++) {
        if(numbersList[i] % 2)
            nums[0]++;
        if(numbersList[i] > (numElements / 2))
            nums[1]++;
    }
}
```

A.9 Exploit Algebraic Identities

A.9.1 Before

```
function beforeFirst() {
    var num = Math.floor(Math.random() * 20) + 1;

    return(Math.pow(num, 2) > 100);
}

function before() {
    var a = Math.floor(Math.random() * 2);
    var b = Math.floor(Math.random() * 2);

    return(!a && !b);
}
```

A.9.2 After

```
function afterFirst() {
    var num = Math.floor(Math.random() * 20) + 1;

    return(num > 10);
}

function afterSecond() {
    var a = Math.floor(Math.random() * 2);
    var b = Math.floor(Math.random() * 2);

    return(!(a || b));
}
```

A.10 Short-Circuit Monotone Functions

A.10.1 Setup

```
var numbersList = [];
for(var i = 0; i < 1000000; i++){
    var num1 = Math.floor(Math.random() *  100000) + 1;
    var num2 = Math.floor(Math.random() *  100000) + 1;
    var pair = [num1, num2];
    numbersList.push(pair);
}
```

A.10.2 Before

```
function before(numElements) {
    var minIndex = 0;
    var minSum = numbersList[0][0] + numbersList[0][1];

    for(i = 0; i < numElements; i++) {
        var thisSum = numbersList[i][0] + numbersList[i][1];
        if (thisSum < minSum) {
            minIndex = i;
            minSum = thisSum;
        }
    }
}
```

A.10.3 After

```
function after(numElements) {
    var minIndex = 0;
    var minSum = numbersList[0][0] + numbersList[0][1];

    for(i = 0; i < numElements; i++) {
        if(numbersList[i][0] < minSum) {
            var thisSum = numbersList[i][0] + numbersList[i][1];
            if(thisSum < minSum) {
                minIndex = i;
                minSum = thisSum;
            }
        }
    }
}
```

A.11 Reorder Tests

A.11.1 Before

```
function beforeFirst() {
    var num = Math.floor(Math.random() * 100);

    if(num <= 15)
        return 1;
    else if(num <= 50)
        return 2;
    else if (num > 50)
    return 3;
}

function beforeSecond() {
    var num = Math.floor(Math.random() * 55);

    if(num == 0)
        return 1;
    else if (num <= 2)
        return 2;
    else if (num <= 5)
        return 3;
    else if (num <= 9)
        return 4;
    else if (num <= 14)
        return 5;
    else if (num <= 20)
        return 6;
    else if (num <= 27)
        return 7;
    else if (num <= 35)
        return 8;
    else if (num <= 44)
        return 9;
    else if (num <= 54)
        return 10;
}
```

A.11.2 After

```
function afterFirst() {
    var num = Math.floor(Math.random() * 100);

    if(num > 50)
        return 3;
    else if(num > 15)
        return 2;
    else
        return 1;
}
function afterSecond() {
    var num = Math.floor(Math.random() * 55);

    if(num > 44)
        return 10;
    else if (num > 35)
        return 9;
    else if (num > 27)
        return 8;
    else if (num > 20)
        return 7;
    else if (num > 14)
        return 6;
    else if (num > 9)
        return 5;
    else if (num > 5)
        return 4;
    else if (num > 2)
        return 3;
    else if (num > 0)
        return 2;
    else if (num == 0)
        return 1;
}
```

A.12 Precompute Logical Functions

A.12.1 Setup

The setup for this rule is not present in this appendix as it is way too large. The *before* function returns the fibonacci number for a given index, which is calculated recursively. The *after* function simply gets the number from an array, which contains all possible fibonacci numbers.

As such, the setup was simply an array which contained every possible fibonacci number, in order of their index, up until the index 1476, which was the maximum allowed, before the program started considering the result to be an infinite number.

A.12.2 Before

```
function before(num) {
    var a;
    var b;
    var c;
    if(num === 0 || num > max_fibonacci)
        return 0;
    else if(num <= 2)
        return 1;
    else{
        a = 1;
        b = 1;
        for(var i = 3; i <= num; i++){
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

A.12.3 After

```
function after(num) {
    if(num > max_fibonacci)
        return 0;
    return fibonacci_array[num];
}
```

A.13 Control Variable Eliminations

A.13.1 Setup

```
var numbersList = [];

for(var i = 0; i < 1000; i++) {
    numbersList.push(Math.floor(Math.random() * 1000) + 1);
}
var numToSearch = Math.floor(Math.random() * 1000) + 1;
```

A.13.2 Before

```
function before() {
    var i = 0;
    var keepGoing = true;
    while(keepGoing) {
        if(i >= numbersList.length)
            keepGoing = false;
        else if (numbersList[i] == numToSearch)
            keepGoing = false;
        else
            i++;
    }
    return i;
}
```

A.13.3 After

```
function after() {
    var i = 0;
    numbersList.push(numToSearch);
    while(numbersList[i] != numToSearch) {
        i++;
    }
    return i;
}
```

A.14 Collapse Procedure Hierarchies

A.14.1 Setup

```
var numbersList = [];
function generateNewList(numElements) {
    numbersList = [];
    for(var i = 0; i < numElements; i++) {
        numbersList.push(Math.floor(Math.random() * 10000) + 1);
    }
}
```

A.14.2 Before

```
function beforeFirst(numElements) {
    generateNewList(numElements);
    return numbersList.length;
}
function beforeSecond(numElements, numIterations) {
    for(var i = 0; i < numIterations; i++) {
        generateNewList(numElements);
    }
    return numbersList.length;
}
```

A.14.3 After

```
function afterFirst(numElements) {
    numbersList = [];
    for(var i = 0; i < numElements; i++) {
        numbersList.push(Math.floor(Math.random() * 10000) + 1);
    }
    return numbersList.length;
}
function afterSecond(numElements, numIterations) {
    for(var i = 0; i < numIterations; i++) {
        numbersList = [];
        for(let i = 0; i < numElements; i++) {
            numbersList.push(Math.floor(Math.random() * 10000) + 1);
        }
    }
    return numbersList.length;
}
```

A.15 Exploit Common Cases

A.15.1 Setup

```
var max_fibonacci = 1476;
var fibonacci_array = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
     987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393,
    196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465,
    14930352, 24157817, 39088169, 63245986, 102334155 165580141, 267914296,
    433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976,
    7778742049, 12586269025, 20365011074, 32951280099, 53316291173, 86267571272,
    139583862445, 225851433717, 365435296162, 591286729879, 956722026041,
    1548008755920, 2504730781961, 4052739537881, 6557470319842, 10610209857723,
    17167680177565, 27777890035288, 44945570212853, 72723460248141,
    117669030460994, 190392490709135, 308061521170129, 498454011879264,
    806515533049393, 1304969544928657, 2111485077978050, 3416454622906707,
    5527939700884757, 8944394323791464, 14472334024676220, 23416728348467684,
    37889062373143900, 61305790721611580, 99194853094755490, 160500643816367070,
    259695496911122560, 420196140727489660, 679891637638612200,
    1100087778366101900, 1779979416004714000, 2880067194370816000,
    4660046610375530000, 7540113804746346000, 12200160415121877000,
    19740274219868226000, 31940434634990100000, 51680708854858330000,
    83621143489848430000, 135301852344706760000, 218922995834555200000,
    354224848179262000000];

function fibonacciSlow(num) {
    var a;
    var b;
    var c;
    if(num === 0 || num > max_fibonacci)
        return 0;
    else if(num <= 2)
        return 1;
    else{
        a = 1;
        b = 1;
        for(var i = 3; i <= num; i++){
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
function fibonacciFast(num) {
    return fibonacci_array[num];
}
```

A.15.2 Before

```
function before() {
    let num = Math.floor(Math.random() * 150) + 1;
    return fibonacciSlow(num);
}
```

A.15.3 After

```
function after () {
    let num = Math.floor(Math.random() * 150) + 1;
    if(num <= 100)
        return fibonacciFast(num);
    else return fibonacciSlow(num);
}
```

A.16 Compile-Time Initialization

A.16.1 Before

```
function before() {
    var a = 10;
    var b = 15;
    var sum = a + b;

    return sum;
}
```

A.16.2 After

```
function after() {
    return 25;
}
```

A.17 Eliminate Common Subexpressions

A.17.1 Setup

```
var list1 = generateNewList(10);
var list2 = generateNewList(10);
function generateNewList(numElements) {
    numbersList = [];
    for(var i = 1; i <= numElements; i++) {
        numbersList.push(i);
    }
    return numbersList;
}
function areEqual(list1, list2) {
    if(list1.length === list2.length) {
        for(let i = 0; i < list1.length; i++) {
            if(list1[i] !== list2[i])
                return false;
        }
    } else return false;
    return true;
}
```

A.17.2 Before

```
function beforeFirst() {
    let string = '';
    if(areEqual(list1, list2)) string += 'The lists are equal...\n';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';

    return string;
}
function beforeSecond() {
    let string = '';
    if(areEqual(list1, list2)) string += 'The lists are equal...\n';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(areEqual(list1, list2)) string += 'The lists are still equal...';

    return string;
}
```

A.17.3 After

```
function afterFirst() {
    let evaluation = areEqual(list1, list2);
    let string = '';

    if(evaluation) string += 'The lists are equal...\n';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';

    return string;
}
function afterSecond() {
    let evaluation = areEqual(list1, list2);
    let string = '';

    if(evaluation) string += 'The lists are equal...\n';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';
    string += 'How about now?\n';
    if(evaluation) string += 'The lists are still equal...';

    return string;
}
```

A.18 Pairing Computation

A.18.1 Setup

```
function ObjectOne() {
    var num1 = 10;
    var num2 = 10;

    this.getNum1 = function() {
        return num1;
    }
    this.getNum2 = function() {
        return num2;
    }
}
function ObjectTwo() {
    var num1 = 10;
    var num2 = 10;

    this.getNums = function() {
        return [num1, num2];
    }
}
```

A.18.2 Before

```
function before() {
    var obj = new ObjectOne();
    var num1 = obj.getNum1();
    var num2 = obj.getNum2();
    var sum = num1 + num2;
    return sum;
}
```

A.18.3 After

```
function after() {
    var obj = new ObjectTwo();
    var nums = obj.getNums();
    var sum = nums[0] + nums[1];
    return sum;
}
```

Appendix B

Data Structures Test Fixtures

This appendix contains the test fixtures for the testing of different data structures to be used on the memoization transformation, as described in Chapter 5. These test fixture pretend to test the performance of arrays, maps and objects, by executing insertions and searches on these data structures.

B.1 Array Test Fixture

```
var memo_array = function (func) {
  var array = [];
  return function () {
    var key = JSON.stringify(arguments);
    if (key in array) {} else {
      array[key] = func.apply("null", arguments);;
    }
    return array[key];
  };
};

var fibo_array = memo_array(function(n) {
  if (n === 0 || n === 1)
    return n;
  else
    return fibo_array(n - 1) + fibo_array(n - 2);
});

fibo_array(1476);
```

B.2 Map Test Fixture

```
var memo_map = function (func) {
  var map = new Map();
  return function () {
    var key = JSON.stringify(arguments);
    if (map.get(key) == undefined) {
      map.set(key, func.apply("null", arguments));
    }
    return map.get(key);
  };
};

var fibo_map = memo_map(function(n) {
  if (n === 0 || n === 1)
    return n;
  else
    return fibo_map(n - 1) + fibo_map(n - 2);
});

fibo_map(1476);
```

B.3 Object Test Fixture

```
var memo_object = function (func) {
  var obj = {};
  return function () {
    var key = JSON.stringify(arguments);
    if (key in obj) {} else {
      obj[key] = func.apply("null", arguments);;
    }
    return obj[key];
  };
};

var fibo_object = memo_object(function(n) {
  if (n === 0 || n === 1)
    return n;
  else
    return fibo_object(n - 1) + fibo_object(n - 2);
});

fibo_object(1476);
```

References

Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. 2008.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

Thoms Ball. The concept of dynamic analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 216–234. Springer-Verlag, 1999.

Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982. ISBN 0-13-970251-2.

David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.27. URL https://doi.org/10.1109/FOSE.2007.27.

CERT-UK. Code obfuscation. https://www.ncsc.gov.uk/content/files/protected_files/guidance_files/Code-obfuscation.pdf, 2014.

Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008.

Christian Sven Collberg, Clark David Thomborson, and Douglas Wai Kok Low. Obfuscation techniques for enhancing software security, December 23 2003. US Patent 6,668,325.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

Martin Fowler. Isoptimizationrefactoring. https://martinfowler.com/bliki/IsOptimizationRefactoring.html, 2004. Accessed: 2018-07-12.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

Google. The google closure compiler. https://developers.google.com/closure/compiler/. Accessed: 2018-07-11.

Jake Gordon. Code incomplete. https://codeincomplete.com/games/.

Matthias Hauswirth and Trishul M Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Acm Sigplan Notices*, volume 39, pages 156–164. ACM, 2004.

Ariya Hidayat. Esprima. http://esprima.org/.

Jscrambler. Jscrambler. https://docs.jscrambler.com/?_ga=2.127463757.1333255579.1531439490-705390441.1528977856], note = Accessed: 2018-07-12.

Bhavnesh Kaalra and K Gowthaman. Cross browser testing using automated test tools. *International Journal of Advanced Studies in Computers, Science and Engineering*, 3(10):7, 2014.

Microsoft. Support for older versions of internet explorer ended. https://www.microsoft.com/en-us/windowsforbusiness/end-of-ie-support, 2015. Accessed: 2018-08-18.

Mozilla. Mdn web docs - cache. https://developer.mozilla.org/en-US/docs/Web/API/Cache, 2018.

PR Newswire. Press release announcing javascript. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html, 1995. Accessed: 2018-07-12.

Selvakumar Samuel and A. Kovalan. A design level optimization approach for functional paradigm software designs considering low resource devices development. *Indian Journal of Science and Technology*, 9(21), 2016. ISSN 0974 -5645. URL http://www.indjst.org/index.php/indjst/article/view/95208.

Net Market Share. Browser market share. https://netmarketshare.com/, 2018. Accessed: 2018-08-17.

Alex Shellhammer. The need for mobile speed: How mobile latency impacts publisher revenue. https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/. Accessed: 2018-06-25.

Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, December 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409374. URL http://doi.acm.org/10.1145/1409360.1409374.

Statcounter. Browser market share worldwide. http://gs.statcounter.com/, 2018. Accessed: 2018-08-17.

IL Stats. Internet live stats. *Pobrano z lokalizacji Internet Live Stats: http://internetlivestats. com (20.02. 2017)*, 2017.

S. Vinoski. A time for reflection [software reflection]. *IEEE Internet Computing*, 9(1):86–89, Jan 2005. ISSN 1089-7801. doi: 10.1109/MIC.2005.3.

Tajana Šimunić, Luca Benini, Giovanni De Micheli, and Mat Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 1-58113-267-0. doi: 10.1145/501790.501831. URL http://dx.doi.org/10.1145/501790.501831.

W3Techs. Usage of javascript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all. Accessed: 2018-06-25.

Niklaus Wirth. *Algorithms+ Data Structures= Programs Prentice-Hall Series in Automatic Computation*.

Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.