An energy-oriented look into memoizability, using static analysis

Author:
Theodor Constantin

Supervisor:
Bent Thomsen

Aalborg University
Computer Science (IT)

This page was left intentionally blank

Computer Science (IT)
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
An energy-oriented look into memoizability, using static analysis

Project Group:
pt105f21

Participant:
Theodor Constantin

Supervisor:
Bent Thomsen

Page Number:
56

Date of Completion:
June 14, 2021

Abstract:

Energy awareness has become a hot topic in software development and computer science in recent years, due to realizations concerning the amount of energy consumed through ICT processes worldwide. Practitioner-oriented research reveals that a recurring problem in adopting energy-saving mentalities and techniques for developers is the lack of supporting tools in this area. In this paper, we go over the development of 'grint', a 'green linter' for JavaScript programs. 'grint' is designed to detect functional purity in JavaScript functions, that can be optimized via memoization, in order to improve the energy consumption of the application. I test the efficacy of 'grint' and memoization, as an energy optimization technique, on two open-source projects by running original and refactored versions, as part of an experimental suite, and comparing their energy consumption. I use Intel's RAPL to perform energy measurements. Measurements show that the memoized versions of the applications had consistently worse energy consumption rates than the original. This suggests that function memoizability is a more complex aspect of JavaScript functions, that is only conditioned by purity and not fully represented by it.

This page was left intentionally blank

Summary

This project ventures to assert the worth of memoization, as an optimization technique, from an energy-aware perspective, in JavaScript applications. Memoization is a program optimization technique that involves recording function return values into a memoization table, such that the values can be retrieved from the table, on ulterior calls with the same set of arguments. This way, function computations can be avoided and replaced by simple table lookups. However, in order to be memoized, functions are required to be pure.

In order for a function to be pure, it must not cause side-effects in the program and its return value must always be the same, for the same set of arguments. Functional purity is innate to subroutines in functional programming languages. However, in an imperative language such as JavaScript, functional purity must be explicitly defined, before being used conceptually in a project such as this one.

In order to facilitate the project, and to put into perspective the meaning of functional purity in JavaScript, I construct a static analysis tool, a linter, that can scan a JavaScript project directory for functions, and assert their purity. In doing so, I define JavaScript function purity as a composite of two perspectives: Independent purity and contextual purity. Independent purity concerns aspects of functional purity that can be decided simply by looking at the body of a function. Independent purity assertion ensures that a function does not, in and of itself, directly perform any operations that cause side effects, is not a higher order function and returns a value. Contextual purity takes into account other functions that are being called from the body of the tested function. A function's purity depends on its functional dependencies. Contextual purity assertion generates a call-graph of functions in a project and propagates purity statuses throughout the graph. The linter implemented in this project that implements this type of functional purity assertion is called 'grint' and classifies functions into three categories: 'pure', 'impure' and 'unknown'. Functions classified as 'unknown' are functions that are not explicitly impure, but cannot be confidently classified as pure either. This is because 'grint' does not currently take into consideration called functions that are not declared within the project.

The linter is used in order to detect safely memoizable functions in two open-

source projects: ‘marked’, a markdown-to-HTML compiler, and ’trianglify’, a library that can generate triangle-based graphical art. All of the memoizable functions found are part of the ’unknown’ category, which emphasizes the incompleteness of ’grint’. In order to assert the possible energy consumption benefits of memoization, I manually memoize the chosen functions, rebuild the project as memoized versions, execute them as part of a test suite and compare their performance with the original versions. Results show that memoization consistently worsened the performance of the applications. While not having investigated the reasons for this, I postulate that the memoization overhead caused by having to process complex function arguments into strings, such that they can be used as keys in the memoization table, is of higher cost than the computations of the functions themselves.

The main takeaway from the results of the experiments is that memoization in JavaScript is a more complex aspect of functions that just purity nad that reckless memoization can worsen the performance of an application, rather than improve it.

An energy-oriented look into memoizability, using static analysis

Theodor Constantin

June 14, 2021

Abstract

Energy awareness has become a hot topic in software development and computer science in recent years, due to realizations concerning the amount of energy consumed through ICT processes worldwide. Practitioner-oriented research reveals that a recurring problem in adopting energy-saving mentalities and techniques for developers is the lack of supporting tools in this area. In this thesis, we go over the development of `grint`, a *green linter* for JavaScript programs. `grint` is designed to detect functional purity in JavaScript functions, that can be optimized via memoization, in order to improve the energy consumption of the application. I test the efficacy of `grint` and memoization, as an energy optimization technique, on two open-source projects by running original and refactored versions, as part of an experimental suite, and comparing their energy consumption. I use Intel's RAPL to perform energy measurements. Measurements show that the memoized versions of the applications had consistently worse energy consumption rates than the original. This suggests that function memoizability is a more complex aspect of JavaScript functions, that is only conditioned by purity and not fully represented by it.

Contents

1 Introduction

Due to the impending problem posed by global energy consumption rates, energy awareness has become an issue concerning more specialized domains of human activity. Specifically, **ICT (Information and Communications Technology)** accounts for over a tenth of the global consumption rates and is expected to double in the next decade [1].

This realization has prompted practitioners and academia to invest in the development and research of energy saving methods and technologies that could reduce the global impact of ICT. This was probably only galvanized by the increasing use and usefulness of mobile and battery-dependent gadgets, which are constrained by energy-related requirements. This increased interest in energy awareness in ICT has given rise to the **Green IT** movement [3], a global endeavour to raise awareness about ICT consumption rates and develop a more energy-conservative culture in IT development. In this thesis, I will refer to **Green Software** [4], a specific branch of **Green IT** that concerns the *software* aspect of ICT.

The complexity of ICT, as a domain of activity, makes it rather difficult to reform, such that it adopts a more energy-conservative attitude. New energy methods and solutions are investigated by academia or in-house R&D specialists, while practitioners are supposed to apply them, as part of development processes managed by companies and enterprises. There are several caveats that impeded this entire process:

- not all **Green Software** research is immediately useful and applicable for practitioners;
- research has shown that practitioners, while interested in **Green Software** ideas, are stumped by the lack of utilities to help them develop more energy-efficient software;
- businesses do not necessarily have an incentive to invest in more energy-aware development processes, especially if development time is significantly impacted by this choice;
- researchers are not necessarily aware of the specific needs that practitioners may have, in terms of energy-aware methodologies and tools; this creates the specific need for **practitioner-oriented research** [5], in order to reveal and understand these needs.

An aspect of **Green Software**, of which development could potentially bring significant improvements to the movement, is the **development of tools and utilities** to aid practitioners in architecting energy-efficient software. Previous research [6–8] has investigated the strength of static analysis tools for source code

in detecting energy-inefficient patterns, and results suggest that these utilities are worthwhile. In this thesis, I venture to investigate the potential of a linter, such a static analysis tool, for aiding the optimization of memoizable functions, in JavaScript applications.

1.1 Problem statement

Linters are static analysis utilities that parse and process source code in order to detect specific patterns for various purposes. The term originates from **lint**, a static analysis tool developed by **Stephen C. Johnson** in 1978, at Bell Labs [9], which was a **C** and **C++** *preprocessor* designed to detect troublesome areas of code, undetected by the compiler, that may cause problems at runtime [10]. Later, **lint** started being used outside of Bell Labs, and the usefulness of the tool prompted the development of other variations for **C** and **C++**. Eventually, *lint-like* tools were developed for other languages and the term became a denomination for static analysis tools that detect *code smells* [8].

Linters are especially useful in the context of dynamically typed scripting languages, that cannot make use of compilers in order to check erroneous areas of code. For example, **ESLint** is a well-known linter for JavaScript, that is nowadays included by default in most IDEs (Integrated Development Environments) and text editors, and can be configured to detect and refactor even developer-defined patterns of code. The ubiquity of ESLint and the possibility to use it both as a development tool and as part of Continuous Integration pipelines makes it a viable tool in the Software development industry. In the case of Python, a comparable scripting language, **PyLint** is the most popular such linting tool, accompanied by multiple other popular choices (*PyFlakes*, *pycodestyle*, *pydocstyle*, *Bandit*, *MyPy*).

Linters are not only used to detect defective code, but also for less functional aspects of code, such as quality attributes. For example, ESLint is often used amongst teams of developers to define source code standards and reject pipelines on which code does not respect the convened upon standards. This is so that repositories are homogenous in terms of coding style. This is to show that the range of uses for static analysis tools is fairly wide and often problems that concern source code can probably see some form of resolution in static analysis.

On this note, I introduce this paper by suggesting a linting tool as a solution for energy inefficient source code. Previous work [11] has documented the reluctance of practitioners to adopt Green Software practices due to the inexistence of usable support tools for writing energy-efficient code. A linter as a solution for *energy-greedy* code patterns has been approached as a topic for research before [6], and I will discuss some of this relevant work in the Related Work section. The novelty of

this piece of research is given by the specific energy optimizable that I choose to approach: memoizable functions.

A paper written by Pinto et al. [12] investigates the potential for energy savings of memoization, an optimization technique consisting in the avoidance of unnecessary function computations by caching results upon the first function call with certain arguments and accessing the cache upon function ulterior calls to that function. The paper's results suggest a significant energy optimization for a majority of the chosen functions. However, the authors manually picked their optimizable functions. In this paper, I detail the development of a linter that specifically detects and refactors memoizable functions, and assess the energy-saving potential of such an optimization technique.

I choose to use JavaScript as the language targeted by the linter, firstly because it has been consistently one of the most popular programming languages [13, 14] and, as such, the contributions of this project should have relevance to a greater base of practitioners. Secondly, JavaScript is an interpreted programming language and such languages have a tendency to be less performant [15]. Optimizations for a slower programming language that has a greater degree of popularity are probably more influential and welcomed.

Thus, I formulate the following research questions, which I will attempt to answer by the end of this paper:

Research questions:

1. What impact can static analysis have on the energy consumption behaviour of a JavaScript application?
2. How can we define purity for subroutines in an imperative programming language, such as JavaScript?
3. What impact does the memoization of pure subroutines have on the energy consumption of an application?
4. How can we build a static analysis tool that detects and refactors memoizable functions?

Before any further discussion, it is worth mentioning that, apart from producing interpretable results suggesting the energy-saving potential of memoization, I intend for this paper to act as a cookbook for static analysis tools of this king, having the guide to creating a linter as one of the major contributions of this paper.

1.2 Related work

In [11], Manotas et al. produce a study detailing the opinions of a number of software engineers in regard to writing energy efficient code. The practitioners

were chosen such that the diversity of experience and domain was maximized amongst themselves. The study reveals, alongside other results, that engineers show some degree of reluctance towards energy-efficient code due to the lack of proper supporting tools that would aid them in producing *green code* at a viable pace. The result suggests that a static analysis tool detecting optimizable code could be a worthwhile endeavour.

Pinto et al. investigate the feasibility of memoization as a way to write more energy-efficient applications in [12]. The authors select a number of *pure functions*, from 3 different Java code repositories, and run and compare optimized and original code on the Android platform. Their results suggest that memoization can help save energy for most of the selected functions, although it can worsen consumption in certain cases. The authors do not investigate further the nature of the functions that do not benefit from memoization.

Given that Java is an Object-Oriented language, the study made by Pinto et al. requires a definition for functional purity within an imperative context, given that pure functions are a concept rather associated with the functional paradigm. The authors cite the work of Yang et al., who study the purity of methods in Java and offer a automatic method for inferring function purity in an object-oriented language [16]. In this study, Yang et al. emphasize the differences between imperative and functional subroutines, and give a set of rules for identifying *pure methods*, and how these might vary.

Couto et al. produce a static analysis tool dubbed *Chimera* in [6], which is designed to detect *EGAPs* (Energy Greedy Android Patterns) in Java code specifically written for the Android platform. The authors comprise a list of some of the most encountered and impactful energy-greedy patterns and construct a static analysis framework to detect and refactor these patterns. Their results show that refactored versions of the applications are consistently and significantly more energy efficient than the original ones.

Nicolay et al. devise an approach to detecting functional purity in JavaScript, based on abstract interpretation [17]. Comparatively to the tool used in this project, their approach is more robust and can detect functional purity even in the case of higher order functions, which is a shortcoming of the linter I make use of. As opposed to their method, I use an Abstract Syntax Tree analysis approach to detecting patterns that denote functional purity. The two approaches are alternatives to each other and their work represents a different method of performing what I attempt in this thesis.

2 The Linter

2.1 The original `lint`

In July, 1978, Stephen Johnson explains in a paper [10] the use and implementation of `lint`, the tool he had been working on within Bell Labs up to that point, that would later be used by a wider base of programmers and that would inspire the creation of other such static analysis tools.

Firstly, Johnson mentions the importance of `lint` as an appendix to the C compiler, rather than as a part of it. While a compiler is required to process input code as fast and succinctly as possible, such that it accurately represents the specification of its language, a linter verifies source code in much more detail, such that it helps detect problems with the code that do not concern the language itself. A compiler is supposed to catch incorrect uses of the language, while a linter searches for patterns that negatively impact development, but that are not necessarily "incorrect." Despite this observation, nowadays, compilers for C and C++ are equipped to give warnings for some of the *code smells* that `lint` was designed to. Nevertheless, linters still exist for most programming languages, as they can help detect specific unwanted patterns of code, whatever these are. Customizability is an advantage given by the auxiliary nature of linters.

In his paper, Johnson mostly describes the functionality of `lint` and whatever blind spots it might have in detecting unwanted code patterns. A casual takeaway from his descriptions is that compromises are to be expected when building such a tool. A static analyser is naturally limited by what is computable (for example, whether a function can ever terminate). Furthermore, a linter is limited by its static nature, and cannot possibly deduce behaviours of the code that are dependent on input data (for example, certain portions of code might never be executed, depending on runtime input). As such, Johnson seems content with implementing linting algorithms to the best possible extent, despite not being able to detect patterns perfectly.

In terms of implementation, Johnson gives a succinct description of the architecture of `lint`: two programs and a driver. The first program includes a C compiler and has the role to syntactically and semantically analyse the input code. The product of this program is saved to an intermediary file. The second program performs a consistency check on the results of the first. The driver acts as a controller for this entire process [10].

Johnson does not give a detailed guide to how a linter is build (in fact, there is no reason for him to know at the time that "linters" would be a class of tools of their

own). However, there are some useful takeaways that I take into consideration:

- any linter most probably requires a compiler or at least a parser of the input language; the linter should not have the responsibility of checking for lexical and syntactic correctness, as dictated by the language specification; the compiler has that role, and the linter analyses the intermediary output of the compiler, which can be an Abstract Syntax Tree (AST); as an energy-relevant example, in [6], the authors write a linter in which they detect energy-greedy patterns by searching ASTs for specific patterns of code
- functions of the linter might not always be completely accurate; complex linters can miss some of the searched patterns, due to limitations of computability and the reluctance to overcomplicate the architecture of the linter itself;
- a linter is used because it suits more specific requirements of developers programming in a certain language; configurability is an advantage, as it increases the degree to which a programmer can better tailor the tool to their specific needs

2.2 `grint`

In my attempt to perform investigations concerning memoization and its energy saving potential, I develop a *prototype* linter specifically designed to detect memoizable functions. As a side note, function *memoizablity* (which, at the moment of writing this, might not be a word in the English language) is a function that can be memoized, without impacting the functionality of the program. Here, I draw a distinction between `purity` and `memoizability`, as there can be functions that are not strictly pure (at least not by the textbook definition), but that can be safely memoized, with some caveats. For example, these can be functions that depend on external variables that are not expected to change during runtime. On the other hand, by the end of this thesis, results will suggest that memoizability can mean more than simple purity, when considering the purpose for which it is usually employed. I will go into theoretical details concerning purity in the *Purity detection* section.

Similarly to Johnson's `lint`, my linter design is comprised of three main components: a parser, a purity checking module and a memoization module. Of the three, the parser is the component that I can safely import from an external source. For this purpose, I choose a well-known and maintained JavaScript parsing library, `acorn.js`. The purity-checker is responsible for detecting pure functions that can be memoized. It generates a list of all of these functions in a chosen root-directory of a JavaScript project. The developer then has the freedom to choose which of the detected pure functions should be optimized, which is then done by the memoization module.

Similarly to how other tools of this kind often have names that are variations of the original *lint*, I name this tool `grint`, an abbreviation of *Green Lint*. In the rest of this section, I will be detailing the constituent parts of `grint`, all the while I explain any necessary theoretical and technical concepts surrounding JavaScript parsing, functional purity and memoization.

2.2.1 Parsing

A minimal explanation for the functionality of a linter is that it is a tool that detects patterns of code. A technique for doing this is to generate an AST from the input source code and traverse it, while trying to catch the wanted patterns. An **abstract syntax tree** is an intermediary representation of the code, that is ensured to have been lexically and syntactically checked for correctness and that is easier to traverse and decorate with further checks and specifications. For example, a type checker will work with an AST, in order to ensure the type-correctness of a program, by decorating nodes of the tree with type annotations and checking that code constructs throughout the AST abide by their type semantics. In compilation processes, ASTs are valuable artefacts that are obtained via parsing.

Parsers are probably the most documented aspect of computer program compilation, with a plethora of supporting tools and practices, such as parser generators, parser construction libraries, paradigm-specific techniques (such as parser combinators, in the functional paradigm) and even complete and integrable parsers for most languages with some minimal degree of popularity. A language as popular such as JavaScript is bound to have some options for parsers available for public use, case in which I did not see a point in constructing my own parsing unit for JavaScript.

JavaScript is formally defined by the ECMAScript Language Specification [18]. However, the language does not have an official, primary interpreter, compiler or parser. Popular for defining functionality within dynamic web pages, JavaScript is executed by web browsers using various engines, specific to each browser (**V8** in *Google Chrome*, **SpiderMonkey** in *Firefox*, **Chakra** in *Internet Explorer*, etc.) [19–21]. However, there are several well-known parsers and compilers for JavaScript developed by communities of volunteers, that are reputable enough to use. One such library is `acorn`.

There is not much to detail upon `acorn`: it is a lightweight JavaScript parser, with one main function that, given a program string, will return a parsed version of that string, or throw an error, given any syntactical error in the input code; the parsing process is configurable, such that the parser accommodates for various versions of the ECMAScript language, different styles of coding and different ways of feeding the input to the parser; the parser can be extended further with existing

plugins. It is notable that `acorn` produces an AST that is modelled as a JavaScript object, that abides by a community-convened standard for JavaScript ASTs name ESTree [22]. This standard came to existence after a Mozilla engineer, working on `SpiderMonkey`, the Firefox-specific JavaScript engine, exposed and documented the internal format used the SpiderMonkey JavaScript AST. This format is officially represented by recursive TypeScript types, which makes tools such as `acorn` easy to integrate within TypeScript programs, that also need to process the ESTree AST, after it is generated. Finally, I should note that ECMAScript continues to receive updates and improvements, while the ESTree format remains supported by a team of volunteers, who claim that ESTree is a community standard that became a *lingua franca* for applications that process JavaScript code [22].

The format of the AST is important, as it is a functional representation of the grammar of JavaScript, without which I myself would have to derive some sort of type library that emulates the ECMAScript grammar. Thankfully, ESTree provides a good enough such library, that provides sufficient information on AST nodes, such as preceding comments or delimiting line numbers for said code constructs. Lastly, the fact that ESTree provides a TypeScript type library for its standard and that `acorn` is written in JavaScript and is integrable in TypeScript programs determined the choice of TypeScript as the programming language for `grint` to be written in.

2.2.2 Abstract Syntax Tree Traversal

The `acorn` parser produces an AST comprised of various types of nodes. The ESTree specification defines these nodes as a collection of TypeScript interfaces that is conveniently available as a `Node` library. Thus, it is easily usable in a TypeScript workspace.

The ESTree interfaces represent the various code constructs specified by the ECMAScript language. They are an implemented representation of the language's grammar. As such, they closely resemble language grammar rules:

- they build upon each other, either by extension or through union types, much like grammar nonterminals may produce multiple other types of single terminals, alternatively; for example, such as the `Expression` type, which is a union of all different expression interfaces in the library;
- they recursively reference each other, much like grammar nonterminals produce series of other terminals and nonterminals, denoting code constructs; for example, `IfStatement` has as attributes a `test` (`Expression`), a `consequent` (`Statement`) and a possible `alternate` (`Statement`).

The ESTree-format AST is basically a large JSON object, of type `Program`, that

contains the entire rest of the program as an attribute. In order to process such a construct, we need to perform some sort of AST traversal method. A common pattern used in processing ASTs is the **visitor pattern**, a design pattern specific to Object-Oriented Programming, meant to avert the implementation of class-specific functionality by having it contained in a `visitor` object and injecting it in specific classes when being processed [23]. The visitor pattern is implemented as follows:

- a `visitor` class implements a set of `visit` functions for various other classes; in the case of an AST, these classes would represent the variety of AST nodes
- these other classes would expose a function that `accepts` a visitor object and calls the visit function upon itself
- the `visitor` object's `visit` functions then implement whatever processing algorithms are needed for each of the classes that they specifically concern.

It is worth noting that the *visitor pattern* is only an option to AST traversal and that various types of tree processing might require specific methods.

Obviously enough, the *visitor pattern* is OOP-specific, while our implemented AST representation is a collection of interfaces that cannot contain functionality unless converted to classes. As I did not want to tamper with the ESTree library, as that might be more work than it is worth, I devised some sort of a *procedural visitor pattern*.

Firstly, I must mention a convenient aspect of ESTree interfaces, which is that they each specify a `type` attribute, that is of a *literal type*. In TypeScript, one can use a language literal (`string`, `number`, `boolean`, etc.) as a type for a variable declaration. Declaring a variable as such, for example with the type `true`, it can only ever have the value `true`. A `literal type` essentially denotes a set of possible values of size 1. In the ESTree library, the `type` attribute on each interface is of a string `literal type`, denominating the variety of interface that it is contained by. For example, the `IfStatement` interface will have `type: "IfStatement"`. This aspect of ESTree interfaces essentially enables us to determine the type of a node object based on what its `type` attribute is equal to (see Listing 1).

```
1  function processExpression(exp: Statement) {
2    if (exp.type === "IfStatement") {
3      // process `if` statement
4      //...
5    } else if (exp.type === "WhileStatement") {
6      // process `while` statement
7      //...
8    } else if (exp.type === "ForStatement") {
9      // process `for` statement
10     //...
11   }
```

```
12 }
```

Listing 1: An example of how the 'type' field helps with typing inside the switch statement; within each branch, 'exp' will be typed differently

As such, based on the `type` attribute of our AST nodes, we can programmatically redirect them towards different methods of processing each node. This leads to the following method of AST traversal:

- I have a general `visit` function, that accepts as parameters an AST node and a function that accepts a node and returns `any` type of value; the function parameter is optional and defaults to a constant `undefined` function (`(n) => undefined`; see Listing 2);
- the `visit` function returns a `NodeMap` (Listing 3), which specifies the type of the node, data returned by the processing function when applied to it, a dynamic `value` attribute, that might contain specific information about some types of nodes, and a collection of other `NodeMap` objects, denoting the node's children; as an honest personal note, this aspect of the visitor function did not offer much utility, for anything other than debugging; my initial plan for this return type was to simplify the AST to as succinct of a structure as possible; however, that was not entirely necessary
- within the `visit` function, there is a rather large *switch statement* that discriminates the parameterized node based on its `type` attribute (Listing 4), and redirects it as a parameter to specialized visitor-functions, that handle specific types of nodes, along with the parameterized function mentioned above (Listing 5 shows an example for the `FunctionExpression` node)
- the parameterized function is used in each of the node-specific visitor-functions on the node object; its return value is stored in the top level of the returned `NodeMap` object; this function is a way to inject functionality in this tree traversal process, such that we can process whatever types of nodes we wish inside this callable parameter, externally from the visitor-module;
- inside the node-specific visitors, apart from calling the parameter function with the node, we possibly store some node data in the `value` attribute of the resulting `NodeMap` (for example, actual values for literals) and then, call the generic `visit` function on child nodes of the currently visited node; we store the resulting `NodeMap` objects from these calls in the `children` attribute of the currently constructed `NodeMap`; it is worth mentioning that some code constructs have optional child nodes specific to them, such as the `else` branch of an `if` statement, represented in ESTree by the `alternate` attribute of the `IfStatement` interface; this needs to be handled, or otherwise the program will throw errors when trying to access attributes on `null` objects;
- at the end of this traversal process, all nodes in a tree will have been tra-

versed, mapped to the resulting `NodeMap` and will have been processed by the parameterized function.

```
function visit(node: Node, func: (n: Node) => any = (n) => undefined): NodeMap;
```

Listing 2: The definition of the 'visit' function

The functionality injected via the function parameter of the visitors can be used to both alter the original AST, or to return relevant data regarding any specific types of nodes. For example, in order to extract all function calls from the body of a function, we `visit` the `FunctionExpression` node corresponding to that function, with a parameter function that pushes `FunctionCall` nodes into an externally declared container. This array can be declared in the same scope in which `visit` is called, and the scope of the parameter function, provided as a lambda, will be able to reference the external array and modify it contents. This can be isolated into a special function of its own and can be generalized to search for multiple types of nodes at the same time, so as to avert unnecessary extra traversals of ASTs (Listing 6).

```
export interface NodeMap {
  type: NodeType;
  data?: any;
  body?: NodeMap[];
  value?: any;
}
```

Listing 3: The NodeMap interface

```
export function visit(
  node: Node,
  func: (n: Node) => any = (n) => undefined
): NodeMap {
  let nM: NodeMap = { type: node.type };
  switch (node.type) {
    case "Program":
      nM = visitProgram(node, func);
      break;
    case "ClassDeclaration":
      nM = visitClassDeclaration(node, func);
      break;
    case "MethodDefinition":
      nM = visitMethodDefinition(node, func);
      break;

    // ...

    case "ExportSpecifier":
```

```
20        nM = visitExportSpecifier(node, func);
21        break;
22    }
23    nM.data = func(node);
24    return nM;
25  }
```

Listing 4: Snippet of the 'visit' function

```
1  function visitFunctionExpression(
2    functionExpression: FunctionExpression,
3    func: (n: Node) => any = (_) => undefined
4  ) {
5    const nM: NodeMap = { type: functionExpression.type };
6    nM.body = functionExpression.id ? [visit(functionExpression.id, func)] : [];
7    nM.body = nM.body.concat([
8      ...functionExpression.params.map((p) => visit(p, func)),
9      visit(functionExpression.body, func),
10   ]);
11   return nM;
12 }
```

Listing 5: The visitor function for the 'FunctionExpression' type node

```
1  export function getVisitByType(
2    type: NodeType,
3    acc: Node[]
4  ): (node: Node) => void {
5    return (node: Node) => {
6      if (node.type === type) acc.push(node);
7    };
8  }
9
10 export function getByType(node: Node, type: NodeType): Node[] {
11   const acc: Node[] = [];
12   visit(node, getVisitByType(type, acc));
13   return acc;
14 }
```

Listing 6: The functions responsible for retrieving specific types of node form an AST

2.2.3 Purity detection

For a function to be pure, it has to respect the following two conditions:

- the function's evaluation solely depends on its provided arguments, and is consistent, such that, for a particular set of arguments, the function's call will always evaluate to the same value;

- the computation of the function does not have any side effects; this is to mean that the function cannot alter any state outside of its own scope.

Functional programming generally prides itself with functional purity as one of its main strong points. Advocates of functional programming will attribute multiple advantages of this programming style to its purity aspect, such as greater ease to reason about the program or greater algorithmic safety, due to the lack of side effects. However, functional languages are intrinsically specified to enforce functional purity, while imperative and OOP languages are not, although they often offer facilities to accommodate a functional style. As such, this these varieties of programming styles might require a more rigorous definition of functional purity.

JavaScript is a multiparadigm programming language, that can be used functionally relatively easily. Nevertheless, JavaScript also has imperative and OOP aspects to it, which means that we need to define functional purity such that it applies to these contexts as well.

A paper written by Yang et al.[24] investigates the concept of *pure methods* in Java, an OOP programming language. The authors remark the fact that OOP languages require somewhat of a redefinition for function purity and devise a method of detecting pure functions in Java. An interesting takeaway from this piece of research is a suggested classification of functional purity [24]:

- **stateless** - conforming to the original definition of purity, it means that the return value of the function does not depend on external variables, other than the provided arguments;
- **stateful** - meaning that such a function returns a value that can be dependent on the state of the object it belongs to.

In another paper involving Yang [16], the authors investigate refactoring methods revolving around function purity in Java. Amongst others, they discuss memoization and suggest three preconditions for a function to be memoizable:

- **purity** - this refers specifically to the lack of side effects; the authors imply in the third pre-condition that the function's return value can actually depend on external values, to some extent;
- **argument immutability** - the type of the arguments should enforce immutability; this is because mutating members of an object argument in Java will mutate the original object passed to the function;
- **limited external dependency** - the authors exclude static fields, public member fields and exposed member fields from the allowable dependencies for the memoizable function's return value; this implies that the value of the function's return cannot possibly be affected by any functions that do not

belong to the encapsulating class; this last point is interesting, as it breaks the original definition of purity, while the authors consider *stateful purity* enough to render a function memoizable.

As a side note, Pinto et al., who investigate the energy saving potential of memoization in [12], use these pre-conditions to identify memoizable functions.

In the case of JavaScript, purity detection is significantly more complicated, given the dynamically typed nature of the language. A strong type system, such as the one Java has, gives more information about the program at compile-time, such that static analysis on the program can be much more fruitful and can make decisions about the meaning of the program with higher certainty. On the other hand, statically analysing dynamically typed code requires more inference and has to take into consideration use cases for the code that might never even happen at runtime, depending on input information.

Considering the natural inaccuracy of a purity detector for JavaScript, in this project I aim to build a helper tool to identify functions that are likely to be pure. More precisely, I build a tool that can detect impure functions, based on the most significant indicators, probably impure functions, based on weaker indicators of impurity and possibly pure functions, through exclusion of the rest. In order to perform this, I first need to decompose impurity detection into smaller, more cohesive modules.

Firstly, we should note that a function can be considered impure if it either directly breaks the rules of purity (no side effects, no external dependencies that might influence its evaluation) or if it calls another impure function. As such, a function's impurity can be determined either independently, or contextually. This determines a two-pronged classification of my impurity checks:

- **independent checks** - these are the types of checks that verify whether a function directly determines its own impurity status; independent checking can detect whether a function is impure solely by analysing its body, without having to know anything about the other functions it calls; for example, if a function attempts to assign a value to an identifier that has not been declared within the body of said function, then we can assert that the function is impure; if a function is not independently impure, that does not imply purity; purity can only be asserted once the contextual check has been run and all function calls within the body of said function are pure; this implies that if a function is not independently impure and it does not call any other functions, then it is pure;
- **contextual checks** - a function's purity status depends on the purity status of the functions it calls within its body; a function can become impure

through no refactoring of its own, but rather just by having one of its calls become impure; this implies that purity detection is a matter of constant workspace/project analysis, in order to contextualize a function's role in determining other functions' purity status; unfortunately, this also implies an immediate limitation of such a purity detector: libraries cannot be included in the analysis, as it might be unfeasible to do so, given the magnitude of such a task; this is especially exacerbated by the fact that functions have to be re-contextualized with each refactor; alas, the contextual check is necessary to form a map of function purity for a particular project.

Contextual checking forms a map of dependency between functions, which is essentially a call graph, while independent checks discover the independently impure functions, from which the impurity status will propagate throughout the project call graph. Whatever functions are not flagged as impure after the propagation process are possibly pure.

In the rest of this section, I will go into more technical details surrounding the independent and contextual purity checks.

2.2.3.1 Independent purity I consider a function to be independently pure if it does not **(1)** directly cause side effects in the program and **(2)** it does not directly refer to external variables in its calculation of the return value. This means that a function's independent purity aspect can be asserted simply by analysing the body of the function, and nothing else. Independent purity does not concern itself with calls to other functions, that may be impure themselves, as this would imply an analysis of dependencies, ergo, code external to the body of the function currently being analysed.

In order to assert independent purity, I separate the analysis into multiple subchecks that concern a single type of impure behaviour, as I explain in the previous paragraph. For each of these, I construct a function that accepts a `FunctionExpression` or `FunctionDeclaration` object and returns a `boolean` value, denoting the whether the function in question passed the respective purity check. The entire set of checks is placed into a collection that is used by the one entry point of the independent purity module to verify each function that it is invoked with, as a parameter. One failed check is enough to mark the function as independently impure. As such, as a small optimization, at the first failed check, the function is asserted as being impure and no other checks are run on it. This creates an incentive to order the check collection based on how computationally heavy the checks are, such that the less expensive ones are run first.

```
1 export function checkIndependent(
2   fExp: FunctionExpression | FunctionDeclaration
```

```
): boolean {
  for (const check of purityChecks) {
    if (!check(fExp)) return false;
  }
  return true;
}
```

Listing 7: The top level check functions that sequentially performs all other checks

The list of independent purity subchecks is, as follows:

- **returnCheck** (Listing 8) - in order for a subroutine to be a function, it must return a value, based on its parameters; a subroutine without a return statement is not technically a function, but rather a procedure; in JavaScript, a function with no return statement evaluates to the `undefined` value, which means that, technically, any JavaScript subroutine is a function and the lack of a return statement in a function does not *per sa* imply impurity; however, it does render the function non-memoizable, because the entire purpose of memoization is to cache possible return values for a function; having no explicit return value defeats the purpose of memoization; as such , for the purpose of this project, functions lacking return statements are to be deemed impure;

```
function returnCheck(fExp: FunctionExpression | FunctionDeclaration): boolean {
  let nM = visit(fExp);

  function pruneNestedFunctions(root: NodeMap): NodeMap {
    if (root.body) {
      const newBody = root.body
        .filter(
          (n) =>
            n.type !== "FunctionExpression" && n.type !== "FunctionDeclaration"
        )
        .map(pruneNestedFunctions);

      root.body = newBody;
    }

    return root;
  }

  nM = pruneNestedFunctions(nM);

  function checkForReturn(root: NodeMap): boolean {
    return (
      !!root.body &&
      (root.body.some((n) => n.type === "ReturnStatement") ||
```

```
25          (() => {
26            for (const n of root.body) {
27              if (checkForReturn(n)) return true;
28            }
29            return false;
30          })())
31      );
32    }
33
34    return checkForReturn(nM);
35  }
```

Listing 8: The check responsible for verifying whether a function contains a 'return' statement

- **assignmentCheck** (Listing 9) - there must not be any assignments within the body of the function done to identifiers that have not been declared within the body of the function; this is the best indicator of side effects, as it is clearly a change of program state, external to the body of the function; in order to perform this check, the body of the function is traversed, such that all assignments are ensured to be done to variables declared within the body of the function up to that point; the parameters of the function are also allowed for assignments, as assigning to them does not mutate the possible original variable passed to the function; assignments to attributes of objects should also be handled; an expression through which an object attribute is accessed (of the format `object.attribute`) is called a *member expression* (of type `MemberExpression` in the AST); in order to ensure that assignments are only made to attributes of locally declared objects or arrays, assignment nodes whose left side represents a `MemberExpression` are ensured to have as the root identifier of this member expression a locally declared variable; in this case, parameters are not to be considered safe for assignment, as assignments to member attributes of parameterized objects in JavaScript carry over to the originally passed variable; as such, assignments to parameter members are possibly side effects;

```
1   function assignmentCheck(
2     fExp: FunctionExpression | FunctionDeclaration
3   ): boolean {
4     let localIds: string[] = [];
5     let paramIds: string[] = fExp.params.reduce((acc, curr) => {
6       return acc.concat(patternToIds(curr));
7     }, []);
8     let sideEffect = false;
9
10    visit(fExp, (node) => {
```

```
    if (node.type === "VariableDeclaration") {
      node.declarations.forEach(
        (dec) => (localIds = localIds.concat(patternToIds(dec.id)))
      );
    } else if (node.type === "AssignmentExpression") {
      if (node.left.type === "MemberExpression") {
        const base = getBaseIdentifier(node.left);
        if (!(base.type === "Identifier" && localIds.includes(base.name)))
          sideEffect = true;
      } else {
        const ids = patternToIds(node.left);
        const mutableIds = localIds.concat(paramIds);
        if (ids.find((id) => !mutableIds.includes(id))) sideEffect = true;
      }
    }
  });

  return !sideEffect;
}
```

Listing 9: The check responsible for catching external assignments

- **higherOrderCheck** (Listing 10) - in our context, a higher order function is a function that attempts to call one or more of its parameters; at compile time, it is not possible to assert independent purity of higher order functions, as the parameterized function can possibly be impure and side-effectual and there is no way of knowing this at compile time; it might be possible to infer the purity of the parameterized function by analysing the calls to the function at hand, throughout the project, to verify whether it is only called with pure functions as parameters; however, this is unfeasible for projects that serve as libraries and expect their functions to be used externally; as such, higher order functions are to be deemed independently impure

```
function higherOrderCheck(
  fExp: FunctionExpression | FunctionDeclaration
): boolean {
  let parList: string[] = [];

  for (const param of fExp.params) {
    parList = parList.concat(patternToIds(param));
  }

  let notHigherOrder = true;

  visit(fExp, (node) => {
    if (node.type === "CallExpression" && node.callee.type === "Identifier") {
      notHigherOrder = !parList.includes(node.callee.name);
```

```
15      }
16    });
17
18    return notHigherOrder;
19  }
```

Listing 10: The check that verifies whether the function calls any of its parameters

2.2.3.2 Contextual purity A function is contextually pure if all function calls within its body are pure. It is important to distinguish this side of functional purity from independent purity because it involves the analysis of the entire containing project of the considered functions.

A function's purity depends on the purity of its functional dependencies. This means that a function can have its state of purity change by having its function dependencies change their state of purity. As such, we can model a JavaScript project, in terms of its functions, as a directed call graph, where nodes represent functions whose state of purity depends on the purity of their children in the graph.

Excluding circular dependencies, the call graph will resemble a sort of tree with multiple root nodes, where the leaves are the most basic functions that are either in-built JavaScript functions or do not call other functions, and the roots are the highest-level functions that are not called internally in the projects. For libraries, these functions would represent the exposed interface of functionality that the library offers. For programs with one main entry point, the call graph would be a tree, where the root node is the entry point of the program. The technical name for the "multi-root tree" structure is a *directed acyclic graph*.

In order to form the call graph, the contextual purity module traverses an entire project recursively, starting at its root directory, and records the names of all JavaScript files in a tree. The files in the tree are then parsed though `acorn`, and the file tree is enriched with lists of `FunctionExpression` and `MethodDeclaration` nodes that are present inside its AST. For the purpose of identification, functions and methods are assigned an identification string of the format `<path>:<class>:<identifier>` for methods and `<path>:<identifier>` for top scope functions. For methods, it is necessary to specify the name of the containing class. I will be referring to these identifiers as *function links*. A current known limitation of this identification method is the fact that it does not take into account the possibility of functions with the same name being declared within disjoint scopes.

For each recorded function, we enrich the file tree further with lists of function dependencies, such that each function has a collection of identifier strings of the functions it calls corresponding to it. These identifiers must now be linked to the

functions they refer to.

The function linking process involves the analysis of imports in each file. If a called identifier refers to an internal import, then I use the source file for that import to produce its function link, and replace the string identifier with it. The same is done for in-file references. Functions that are external imports, such as from libraries, and in-built functions are marked with the special character `?`, to denote this.

The module takes into consideration a list of in-built JavaScript functions, and whether they should be considered pure or impure. Another limitation of this method arises here, when considering functions on JavaScript arrays and objects. Should there be calls to an in-built method on arrays, such as `push` or `concat`, there are cases in which it is unsure whether the object on which these methods are called is a JavaScript array or a custom object, that happens to contain a function with the same name as an array function. For example, if a function calls `concat` on a parameter (`parameter.concat(arg)`), it is not possible to know whether the parameter is an array or a custom object. This fact brings doubt to the purity status of the `concat` function. Inference by analysing the uses of the containing function throughout the project might raise confidence in regard to the type of its parameter, but if the function can be used externally, then we cannot make any confident assertions in this regard.

After the linking process, the file tree is converted into a list of objects containing the function link for each function in the project, along with an array of function links denoting each function's dependencies and a field denoting the purity status of the function (`pure`, `impure` or `unknown`). By default, functions are classified as `unknown` in terms of their purity status. This list is then used in the **impurity propagation** process, which follows the following algorithm:

1. the entire function list is evaluated by the independent purity module; this will classify a portion of the original list as `impure` functions;
2. the `impure` functions in the list are then selected and, out of the other functions, that are still marked as `unknown`, it is verified which ones call any of the impure functions;
3. `unknown` functions that call `impure` functions are marked as `impure`;
4. the previous two steps are repeated until there no more `unknown` functions that call any of the impure functions.

A similar process can be applied to propagate purity throughout the list of functions, although it will flag significantly fewer functions, if any. This is because, purity propagation starts with functions that do not make any function calls and are not independently impure, which tend to be fewer. These are marked as `pure`. Propagation then continues by marking `unknown` functions that only call `pure`

functions as `pure`. This repeats until there is no more `unknown` functions that satisfy the condition.

Functions that call external or in-built functions remain marked as `unknown`, because of the reasons specified earlier. To reiterate, we cannot with confidence declare these functions as pure, because of possible naming overlaps with custom functions, in the case of in-built calls, or unknowable purity status of external functions, in the case of external library calls. As such, I decide that the memoization of these functions should remain a choice of the developer. Nevertheless, the linter will inform the user of the list of function declared throughout the project, alongside their asserted purity status.

2.2.4 Automatic reformatting

At the moment of writing this thesis, automatic reformatting in `grint` is subject to future development and, arguably, it is not extremely useful in the current state of the linter. I will explain why this is after discussing how memoization is currently done, for the sake of the experiments, and how I would automate it.

Memoization is done simply via a function that takes another function as a parameter and returns a memoized version. Courtesy to an article on JavaScript memoization written by Joseph Chege [25], we have the function in Listing 11 which instantiates an object as the memoization table and returns a function that checks whether the object contains a previously calculated result; if so, it returns it; otherwise, the function being memoized is run with the arguments of the return function; before returning this results, it is first inserted into the instantiated object, as a member field, with a stringified version of the argument array as the key, and the return value of the memoized function as the value. For the sake of monitoring and debugging, I added a memoization counter that keeps track of the number of memoization table hits for each run of the program. Knowing how often values are being retrieved from the memoization table helps us assert whether memoization is actually helping avert repeated computations in the first place.

```
1  let memoCounter = 0;
2
3  const getCounter = () => memoCounter;
4
5  const memoize = (func) => {
6    const results = {};
7    return (...args) => {
8      const argsKey = JSON.stringify(args);
9      if (!results[argsKey]) {
10       results[argsKey] = func(...args);
11     } else {
12       memoCounter++;
```

```
13      }
14      return results[argsKey];
15    };
16  };
17
18  module.exports = { memoize, getCounter };
```

Listing 11: The memoization function, taken from Joseph Chege's article

In order to memoize a function, we initialize it using the `memoize` function, with the initial lambda as a parameter. If the original function is not declared as a `const` and initialized with a lambda, but rather uses the special `function` syntax, we need to refactor it to a `const` variable, refactor the body and parameters of the function into a lambda, and initialize the `const` with the memoized version of the resulting lambda. This entire process implies that we need to import the `memoize` function into each file we use it in. The way this is done depends on the version of ECMAScript the project uses, as ES6 introduced a new import and export syntax. As such, projects written in ES6 may use the newer import syntax, while older projects will not. As such, memoization refactoring will depend on the version of ECMAScript used in the project. A solution to this would be to give the developer the ability to configure the linter, via a configuration file placed in the root directory of the project, similarly to how other linters can be configured. This can expose the functionality to specify the version of ECMAScript that the project uses, which I can make use of in order to refactor in a version aware manner.

However, the language version issue is not the only aspect impeding automatic refactoring. A project may make use of multiple other linters and static analysis tools for various purposes. One such use is often code style standardization, which checks whether code abides by certain style standards, according to some sort of configuration. ESLint is a well-known example of this. Different configurations for ESLint may denote styles of code that are incompatible with each other's configurations, such that there is no style that satisfies both configurations. This means that, any automatic addition of code will have to know of the ESLint configuration beforehand, in order to produce code that is compatible with the style configuration. Otherwise, there is a risk that automatic refactoring will determine a project to be rejected by possible pipelines that incorporate ESLint into them. This means that something like `grint` cannot properly perform automatic refactoring without the risk of incompatibilities with other project related tools. Nevertheless, the developer can be given the choice to automatically memoize functions, although the effort of marking functions as automatically memoizable is arguably comparable to the effort needed to just memoize the functions.

A first solution for automatic refactoring would be to work directly with the AST,

modify it and then recompile it to code. Once a marked function is found, the AST node corresponding to it can be processed as I discuss above, such that the function becomes a constant initialized with a lambda. The lambda node can then be inserted as a parameter node within a function call to the memoization function. Lastly, we need to include an import statement node in the main scope of the file we are processing. This would correspond to the body of the root node of the AST. It is also necessary to include the file exporting the memoization function in the root directory of the project. A second solution would be to directly modify the code string in the original file, such that we append the necessary code to the functions' expressions in order to memoize them. At the same time, we would need to append the necessary import statement for the `memoize` function at the beginning of the file.

The main advantage with the AST solution is that refactoring is more robust and typed, such that we do not have to consider strange code layouts when processing the AST. As for modifying the file directly, we would essentially need to write a mini-parser for function expressions, such that we can handle various code layouts. On the other hand, refactoring the AST would require us to also include a code generation library that processes ESTree ASTs back into JavaScript code. While researching the means to do this, I happened upon an open-source library called `astring` [26], that promises to do exactly that. However, it generally seems information about converting the ESTree specification back into the ECMAScript code that it represents is scarce.

As a final feature to the automatic refactoring module, the developer should have the choice of which functions are to be memoized. This is because, a majority of memoizable functions that are potentially most impactful, will be marked as `unknown`, in terms of their purity status, as discussed in the previous section. Some other development tools for JavaScript make use of comments in the source code in order to embed otherwise meaningful code in the JavaScript document. A popular example of this is **JSDoc** [27], a markup language used to embed program documentation directly in the JavaScript source code. IDEs will then make use of JSDoc comments in order to display more readable tips and information regarding a program's or library's functions, variables or types. As another example, some IDEs, such as **JetBrains' WebStorm** [28], will use commented annotations in order to mark portions of code in order to override functionality of their embedded linters (an example of this is when the developer is technically breaking a linter rule in the code, but nevertheless needs to do so). Similarly, `grint` can also make use of commented annotations before functions, to mark them as memoizable (for example, something similar to `>>grint:memoize`). Arguably, the linter may also throw errors and give warning s when memoization is attempted for `impure` or `unknown` functions,

respectively. The AST generated by `acorn` gives access to comments prefacing code nodes, which makes the annotation feature relatively facile to implement.

To conclude this section, automatic refactoring, while implementable in different ways, comes with several impediments that can make it ungeneralizable and impractical. As for its usefulness within this project, manual memoization eventually turned out to be quite an easy task, that did not require automation.

2.2.5 Further development

Apart from automated refactoring, there are several other points of improvement for a tool such as `grint` that can elevate its usefulness from a practitioner's perspective. In this section, I will be discussing some of them, alongside implementation possibilities.

2.2.5.1 Configuration A strong point of some of the static-analysis-based tools surrounding the JavaScript development stack is their configurability. Various types of projects might require variations in the way some tools interact with the code. For example, I have already discussed the issue regarding ECMAScript versions: current projects may use different versions of ECMAScript, such that it is often impossible to generalize tools that can properly interact with all versions. As such, linters and other auxiliary tools might require different programmed behaviours for different versions of the language. This can be specified via the `package.json` file that is found in the root directory of the project at hand. In other cases, the tool may prefer to have a specialized configuration file. For example, **Prettier**, an automatic code formatter for JavaScript [29], may make use of a `.prettierrc` configuration file in the root directory of a project, that will specify the style in which the code is to be formatted. Similarly, **Babel**, a JavaScript compiler (and often an alternative to `acorn`), usually used for translating JavaScript code to different versions of ECMAScript, for compatibility reasons, makes use of a configuration file, that specifies compiler options. Comparatively, I provide the `acorn` parser with its compilation options when the main `parse` function is called on a code string.

Likewise, `grint` can also make use of a main configuration file in order to expose some options to the developer:

- **automatic refactoring** - if implemented, automatic refactoring should be a toggleable option, such that the developer has control over the memoization process;
- **ECMAScript version**- seeing as `acorn` itself requires this specification and `acorn` is part of `grint`, it would be necessary to expose this bit of configuration

to the user of `grint`, rather than just setting the `acorn` ECMAScript version to the latest possible one

- **other parser options** - other than the language version, `acorn` also exposes a set of other option for the developer to configure the parsing process with; for example, the input code string can be parsed as a `script` or as a `module`, and this is specified in the parser options with the denomination `sourceType`; this option can be exposed as a `grint` option as well; however, in this particular case, the option applies per file of source code; anecdotally, as far as I have been able to observe in the JavaScript repositories that I considered for this project, projects can be generally parsed using the `module`; this may mean that, the number of outstanding files that might require a different configuration is low enough, such that we can afford to specify a set of parser options for each of these files; the rest of the files in the project would be parsed using a one-time specified set of options; the main conclusion of this entire point is that `grint` configuration should offer the possibility of *per-file* configuration
- **excluded subdirectories** - a project directory may contain subdirectories or files that are not to be considered project source code; furthermore, a root directory for a repository may contain multiple projects; as such, it is necessary that `grint` has a way of specifying which subdirectories or even files are to be excluded from the linting process; this is actually currently done, in a non-configurable way, such that the `node_modules` directory, in which external libraries are installed via the **Node Package Management (NPM)** system, is not traversed by the tool; this is also done for directories of which name begins with a period (`.`), which is a convention originating from Unix-like systems denoting hidden files or directories, that usually contain configuration and meta-information; this part of the linter relies on a list of predicates that check whether a directory should be excluded, which are run on each subdirectory of the root of the project; this can be extended to be configurable and check whether directory names match certain patterns specified in the configuration file; this is reminiscent of the way `.gitignore` files work, with the `git` version management system [30]
- **linter options** - a released version of the linter should have some degree of interaction with the user, via the console; similarly to how C/C++ compilers such as `clang` or `gcc` give meaningful warnings and errors regarding the compiled program, `grint` could also give out errors, when trying to memoize `impure` functions and warnings, when trying to do so with `unknown` functions; linter insights such as these should be toggleable;
- **pure identifiers** - as the developer of `grint`, I cannot confidently make any assertion of purity for functions used in a project, that are not also declared in that project; at most, functions that call such functions are to be deemed

`unknown`; however, if a developer using `grint` has the insight that some of these function identifiers are to be considered universally pure within the project, then they should have the freedom to do so; for example, if the developer knows that the `JSON.parse` function, which parses a `JSON` string into an object, is not otherwise overridden by an impure construct

2.2.5.2 Degrees of memoizability and function complexity analysis As results will show, memoizing functions found to be pure in our chosen repositories increases the overall energy consumption of the program, rather than reducing it. I will go into more details regarding this result in the discussion section of this thesis, but I suspect that the main reason for the lack of improvement is that the processing keys and values for the memoization table is too expensive in the cases where these data are too complex. This will probably happen for functions that have a more complex set of arguments, either a larger number of arguments or more complex objects as arguments or a combination of both factors. When values are memoized, the arguments of the function call being memoized are treated as an array of values and the array is converted to a string, being afterwards used as the key based on which the value will be found in the memoization table. The larger this key is, the greater the overhead of the stringifiation needed to have processed it.

In their work on memoization of Java functions, Pinto et al. [12] make it a point to select methods that specifically have primitive types as arguments. It is specifically because of comparison overhead that they do this. In my case, the lookup overhead is negligible, as I use a JavaScript object as a memoization table, and JavaScript objects are hashes with a constant lookup time [31]. However, the issue of overly complex memoization keys is nevertheless the same between my thesis and their paper. In order for memoization to be a viable optimization technique, its means of implementation must be less expensive than the unmemoized alternative. In [12], Pinto et al. perform a comparison of energy savings for multiple functions when memoized. A personal observation that I made while reading their article was that, amongst some of the worst performing instances of memoization was the function that had a list object as a parameter, alongside a string, which are an arguably complex pair of parameters.

Given this issue, the main takeaway is that there is more to memoizability than just functional purity. The purity aspect of a function denotes whether the function **can** be memoized. However, memoizability should also denote whether a function **should** be memoized, such that the application benefits in terms of performance. This would happen if the overhead added through memoization is overall less costly than performing the actual computation associated with the function in

question. As such, a possible further development for `grint` would be some form of memoizability analysis, such that the developer knows (at least roughly) to what extent a function is worth memoizing.

Calculating a function's degree of memoizability would involve **(1)** calculating the approximate overhead of a memoized function call and **(2)** calculating the cost of executing the function in question. For **(1)**, the overhead score of a memoized function will have to be approximated based on the parameters of the function. The simplest way to do this is to just use the number of parameters as the overhead score. A more accurate way would be to attempt to infer the types of the parameters and incorporate that information into the overhead score. However, as long the method relies on static analysis to assert memoization overhead, it will, at best, produce an approximation rather than an accurate calculation.

For **(2)**, we can assert function complexity relatively more accurately. A simple way to perform such an analysis would be to traverse a function's AST and simply count the number of operations it contains in its body. While not very accurate, because operations, such as function calls and assignments, will differ in complexity and performance cost, this method might be a good initial proxy for relative complexity amongst functions. A more advanced version of this method would be to calculate function cost based on the cost of functions it calls in its body, relying on a project call graph, similarly to how I perform contextual purity analysis in `grint`.

In order to determine memoizability, the final step of this process would be to determine a relationship between memoization overhead and function complexity, and determine which threshold of this ratio separates memoizable functions from non-memoizable ones. We can call this ratio a **degree of memoizability**. Automatically calculating this memoizability score could provide us with a hierarchy of functions, ordered by memoizability, with an arguable level of accuracy. However, this is a barely conceptualized idea and a possible subject for future work.

3 Experiments

The main issue around this project revolves is the overall worth of memoization as an energy optimization technique in JavaScript. In order to study this, I built a tool that automatically finds functions that can be memoized, a linter that maps the network of functions in a project and asserts which ones are `pure`, `impure` or not classifiable as either of these categories confidently (`unknown`).

While developing `grint`, I tested it on an existing repository of code, namely `phaser` [32], a web game development engine written in JavaScript. This meant that, once I had enough of `grint` developed, I could run it on the `phaser` project and check whether it could assert purity correctly, on the scale of a larger project. This method of testing also revealed some shortcomings and bugs that needed to be fixed, in order for the linter to work correctly: badly handled AST nodes, the lack of error messages, the lack of support for functions declared with the same identifier, in the same file. Once the development of `grint`, aided by `phaser` as a subject for analysis, was sufficiently complete to detect pure functions properly, it could be used on other existing repositories and discover potentially memoizable functions. I did not take the refactoring step with `phaser`, because it is a complex game development library that would be rather difficult to construct test cases for and run them in a controlled manner, such that I can measure performance metrics.

The two repositories that I chose to perform my experiments on are `marked` and `trianglify`. `marked` [33] is a JavaScript library that parses `markdown`-formatted text into `HTML`. `trianglify` [34] is a utility that produces triangle-themed graphical art, based on a predefined configuration. The repositories can be found on GitHub and I came upon them while using **Awesome Open Source** [35], a website containing data on over software 370.000 projects, that can be filtered based on various keywords and technologies. The main criterium for choosing my test subjects was the ability to upscale input data, such that I can increase the computational strain upon these libraries. This way, I can read performance metrics at various levels of computational load for the libraries and potentially reveal greater discrepancies between the memoized version and the original one at greater workloads. From this perspective, `marked` can be used with larger and larger markdown files. As for `trianglify`, the configuration settings needed to initialize the art generator can be scaled up such that it produces more complex and higher resolution images.

For each of the repositories, I wrote a small test suite that produces a file of performance metric readings (time and energy). The entire experimentation process, for each project, conforms to the following steps:

1. Each project is analysed with `grint`, which will produce a list of functions, classified as either `pure`, `impure` or `unknown`
2. Of the `pure` functions, I browse the list of `unknown` functions, and select the ones that are pure;
3. All selected functions are memoized and the projects are rebuilt; this is typically done by running the `npm run build` command, which will attempt to execute the `build` task, as defined in the project; as such, this is not a universal solution and, if not already implemented, some other method may be required to build the project; the building process will produce a compiled or bundled version of the project, possibly with several other files, in a reserved folder
4. Within the test suite, I install the original version of the project using `npm`; this will download the library in the `node_modules` folder, in the directory of the test suite;
5. The library folder will then be duplicated and, in the second copy, I replace the project files with the ones produced by building the application; this version of the project will be renamed, to reflect that it is the memoized version, and will be imported inside the test suite as such
6. The test suite is then executed, which will produce a file of metric readings.

In the rest of this section, I explain the experimental methodology relating to the test programs and give technical details about the tools and environment used for running the experiments.

3.1 Methodology

The test suites used for the two repositories are similar in their layout, the only aspect differing being the input data fed to the two libraries. Both versions of a library, original and memoized, are imported at the beginning of the file, alongside the input data, which is provided from external files for both libraries. Each version of the library is run on a piece of the input data, for a given number of iterations. The final results are based on 100 iteration executions. Anecdotally, 100 iterations provided consistent measurement data, without a large number of outliers. However, for a more statistically robust process, as subject for future work, a method for deciding how many samples will produce a statistically significant set of data may be used, such as the application of Cochran's formula [36]. This entire process is then repeated for the rest of the input data. The input data for each library is represented by 5 entries, increasing in computational workload. During each iteration, the test suite registers execution time and RAPL-measured energy in an array, which is saved as a string in an external file at the end of the test suite execution. This is done by reading a timestamp and the RAPL energy consumption

level before and after the execution of the library functionality, and calculating the difference between the two values.

In the case of `marked` (Listing 13), the input data is represented by 5 `markdown` files, of increasing size. At the beginning of each of the corresponding test cases, a file is read as a string and provided to the main function of `marked`, the `parse` function, which will return an `HTML` string.

In the case of `trianglify` (Listing 12), a program execution produces graphical art in the form of an `HTML` canvas, which can then be converted and saved as a `PNG` file. This is based on a predefined set of options, that produce variations in the resulting graphical art, in terms of colours, shapes, resolution, shape sizes, etc. Amongst these options, a randomization seed can be provided, such that any execution of `trianglify` will be identical, given the same seed. This is important, because we need identical test cases in order to have accurate comparisons between the two versions of the library. As such, the input data for `trianglify` is a set of five configurations, with the same seed, and increasing image resolution settings, acting as the scaling factor for the execution workload.

```
const fs = require("fs");
const tri_memo = require("trianglify-memoized");
const tri_norm = require("trianglify");
const performance = require("perf_hooks").performance;

const inputs = require("./inputs.json");
const iterations = 100;

const results = [];
let test_memo, test_norm;

// warmup phase
test_memo = tri_memo().toCanvas();
test_norm = tri_norm().toCanvas();

// iterating through the inputs
for (const input of inputs) {

  let pathResults = {
    path: input.id,
    memo: [],
    norm: [],
  };

  // iterating measurements for the same input
  for (let i = 0; i < iterations; i++) {
    const timeA = performance.now();
    const raplA =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")
    test_memo = tri_memo(input.options).toCanvas();
    const timeB = performance.now();
    const raplB =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")
    test_norm = tri_norm(input.options).toCanvas();
    const timeC = performance.now();
    const raplC =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")

    // registering individual measurements
    pathResults.memo.push({ time: timeB - timeA, rapl: raplB - raplA });
    pathResults.norm.push({ time: timeC - timeB, rapl: raplC - raplB });
  }

  // registering all measurements for the input
  results.push(pathResults);

  const fileMemo = fs.createWriteStream(`./outputs/memo-${input.id}.png`);
  const fileNorm = fs.createWriteStream(`./outputs/norm-${input.id}.png`);
```

```
46
47    test_memo.createPNGStream().pipe(fileMemo);
48    test_norm.createPNGStream().pipe(fileNorm);
49  }
50
51
52  // saving the results in the same directory
53  let file = fs.createWriteStream("./results.json");
54  file.write(JSON.stringify(results));
```

Listing 12: The test file for trianglify

```
const marked_norm = require("marked");
const marked_memo = require("marked-memoized");
const fs = require("fs");
const performance = require("perf_hooks").performance;

const inputPaths = [
  "./inputs/1.md",
  "./inputs/2.md",
  "./inputs/3.md",
  "./inputs/4.md",
  "./inputs/5.md",
];

const iterations = 100;

const results = [];
let test_memo, test_norm;

// warmup phase
const warmup_input = fs.readFileSync(inputPaths[0]).toString();
test_memo = marked_memo(warmup_input);
test_norm = marked_norm(warmup_input);

// iterating through the inputs
for (const path of inputPaths) {
  const input = fs.readFileSync(path).toString();

  let pathResults = {
    path,
    memo: [],
    norm: [],
  };

  // iterating measurements for the same input
  for (let i = 0; i < iterations; i++) {
    const raplA =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")
    const timeA = performance.now();
    test_memo = marked_memo(input);
    const raplB =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")
    const timeB = performance.now();
    test_norm = marked_norm(input);
    const raplC =
        fs.readFileSync("/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj")
    const timeC = performance.now();

    // registering individual measurements
```

```
46      pathResults.memo.push({ time: timeB - timeA, rapl: raplB - raplA });
47      pathResults.norm.push({ time: timeC - timeB, rapl: raplC - raplB });
48    }
49
50    // registering all measurements for the input
51    results.push(pathResults);
52  }
53
54  // saving the results in the same directory
55  let file = fs.createWriteStream("./results.json");
56  file.write(JSON.stringify(results));
```

Listing 13: The test file for marked

```
1  "options": {
2      "width": 200,
3      "height": 200,
4      "cellSize": 25,
5      "variance": 1,
6      "seed": 1,
7      "xColors": "Blues",
8      "yColors": "Purples",
9      "strokeWidth": 0
10 }
```

Listing 14: Example of a trianglify configuration

3.2 Tools and setup

The experiments, in their entirety, were run on a Linux machine, running **Ubuntu** (version 20.04.1 LTS). `grint` was developed in TypeScript and, in order to execute it, I used the `ts-node` utility [37], which is a tool used for executing TypeScript code without first precompiling it to JavaScript. The test suites were run simply using `node`, as they are written directly in JavaScript.

In terms of hardware, the machine on which the experiments were executed is an ASUS laptop, with the following hardware characteristics:

- processor - Intel Core i7-4710HQ 2.50GHz, Haswell
- memory - 12GB, DDR3, 1600MHz, split into two modules of 4GB and 8GB

The machine exposes an SSH server on the local network it is connected to, such that any development and experiment executions can be done remotely, from a different machine. This is in the hope that any reduction of activity on the main experiment machine will work towards more accurate energy readings. However, this choice is only driven by personal intuition, as I am unsure to what extent this method increases energy reading accuracy.

Energy consumption is measured using Intel's RAPL, a utility that keeps track of energy consumed by different power domains of the computer, since it last booted. I discuss RAPL in more detail in the following section. However, it is important to mention here that RAPL is a hardware feature of newer Intel processors. As such, the experiments would only function on a machine that meets this criterium.

3.2.1 Intel's RAPL

The **Running Average Power Limit (RAPL)** is a hardware utility implemented in Intel processors, at least as new as the Sandy Bridge version (2011), that measures energy consumption for a set of different computer consumption domains and can be used to limit consumption for specific domains [38–41].

RAPL primarily relies on a specific type of processor registers called **Model Specific Registers (MSR)** [41, 42]. MSRs are usually used for monitoring and controlling aspects of the computer hardware. In this case, RAPL MSRs constantly retain information about the energy consumption RAPL-monitored domains. These domains are:

- **package (PKG)** - the entire CPU socket
- **power plane 0 (PP0)** - CPU cores
- **power plane 1 (PP1)** - on-chip graphical processing unit
- **DRAM** - dynamic random access memory

These four domains each have their own set of MSRs, that expose a variety of information to the user (energy status, minimum and maximum power, performance impact of manually limiting power for specific components, relevant hardware information) alongside registers that enable the power limiting mechanism. It is worth noting that the DRAM energy domain is supported by the RAPL mechanism starting with Intel processors produced in 2013 (Haswell) or newer.

Khan et al. conduct a study in [38] where they investigate the overall accuracy of RAPL. While having found some specific shortcomings with the tool, they asserted that Intel's RAPL is sufficiently accurate to be trusted as a component-specific energy measurement tool. Simultaneously, papers surrounding the area of Green Software that perform some sort of software energy-related benchmarking have been using RAPL as their main tool of measuring energy [15, 43, 44], which supports RAPL's reputation as a worthwhile utility.

It is worth noting that, due to the fact that RAPL relies on MSRs as a mode of interaction with the user, sing RAPL requires the machine to run a Unix-based operating system (such as iOS or a variety of Linux). In a previous project [5], I discovered first hand that it is impossible to access the RAPL MSRs using the

Windows operating system, because Windows simply does not have access to model specific registers, while Unix-like systems do [45]. This explains my choice for Ubuntu as the operating system for the machine running the experiments.

Accessing RAPL MSRs is a simple task, as information from MSRs can be read from their respective files, in the Linux system directories. These files are updated each time RAPL produces new readings. This happens sequentially, amongst the different domains and, as such, given a program that reads all of them consecutively, there is a possibility of dirty reads, when values for different domains are read that belong to different RAPL update waves.

4 Results

The experiments show clear increases in energy consumption in the memoized versions of the libraries. In all 10 different tests cases, the original version was more energy efficient than the memoized versions. In the first several runs of the experiments, the memoized versions of the libraries consistently showed increased outliers in their first execution of the entire test suite, in terms of both energy consumption and execution time. This prompted me to implement an initial warmup execution for both libraries, which removed the initialization outliers. Furthermore, the memoized libraries are stateful, such that they are not reset in between executions. Because of this, memoization tables are kept from one test case to the next, which will help subsequent test executions by having them make use of previously computed values. However, even with these adjustments and advantages of memoization, the refactored libraries were in all cases both slower and less energy efficient than the original versions.

For `marked`, the energy consumption readings, as shown in Table 1, while consistently larger for the memoized version relatively to the original, the results were arguably subtle. The difference between the average measurement for the first test case, with the smallest workload, was of `0.0025J` in favour of the original version of the library. For the last test case, with the greatest workload, the difference was of `0.4603J` in favour of the original version.

Table 1: RAPL energy reads (Joules) for `marked`, at the different workload levels, for both the original (`norm`) and memoized (`memo`) versions

Workload	Average (`norm`)	Average (`memo`)	Median (`norm`)	Median (`memo`)
1	0.0374	0.0399	0.0277	0.0393
2	0.2170	0.2640	0.2030	0.2404
3	0.9014	1.0516	0.8711	1.0124
4	4.1217	4.6815	3.9611	4.4638
5	4.3037	4.7640	4.2673	4.6313

For `trianglify`, results are more pronounced, as seen in Table 2. The average execution of the first test case is approximately four times more energy consuming for the memoized version of the library then the original, at `0.8517J` as compared to `0.2127J`. For the test case with the greatest workload, the average execution was `25.6` times more energy consuming for the memoized version of the application than the original.

Another aspect of the experiments made more obvious by the `trianglify` results is the increasing discrepancy between the average and the median results. This is consistent in the `marked` results as well, but far more noticeable for `trianglify`. As I note in a previous report [5], with RAPL readings, longer running experiments introduce greater chances for outliers in the resulting measurements. Given that `trianglify` provides a more expensive piece of functionality that should typically take longer to execute, the chances for reading outliers in the `trianglify` results are higher.

Table 2: RAPL energy reads (Joules) for `trianglify`, at the different workload levels, for both the original (`norm`) and memoized (`memo`) versions

Workload	Average (`norm`)	Average (`memo`)	Median (`norm`)	Median (`memo`)
1	0.2127	0.8517	0.1988	0.8143
2	0.6986	5.1638	0.6125	4.5659
3	1.5752	18.2349	1.3623	16.5428
4	2.7348	45.7663	2.4445	44.6527
5	4.6214	118.5109	3.7490	97.9351

After obtaining the measurements, the two data sets for each test case, for the original and the memoized applications, was processed using **Wilcoxon Signed-Ranks Test**, in order to assert whether the two data sets are statistically different [46]. The null hypothesis of the test dictates that the medians of the two datasets are equal [46]. By using a calculator for the Wilcoxon Signed-Ranks Test,[1] with a significance level of `0.05`, I reject the null hypothesis for all datasets, meaning that the differences between the original and the memoized versions are indeed statistically significant.

4.1 Discussion

Despite memoization itself being successful with helping avert recalculations of previously computed function calls, results show that the overhead added through memoization is greater than the cost saved by avoiding some recomputations. This implies that there is a cost to memoization that has to be surpassed by the alternative in order for this optimization technique to be viable. In this project, I constructed a tool that enables me to more easily detect pure functions that can be memoized, without altering the functionality of the overall program. However, I

[1] https://www.socscistatistics.com/tests/signedranks/default2.aspx

mostly took a blind approach when selecting the functions to be memoized, without taking into consideration their complexity or the magnitude of their arguments. Furthermore, a practitioner will have more insight into their own project, such that they can know what functions will be called considerably more frequently and will have the greatest impact when memoized. Obviously enough, the blind approach, primarily taking into consideration purity, is not enough to select memoizable functions, and purity is only one aspect of a more complex problem.

The main takeaway points resulting from the experiments are the following:

1. **There is more to memoizability than purity** - While purity is certainly a necessary precondition for functions to be memoized, memoizability implies that a function should also be worth memoizing, such that it produces performance improvements in the application. In this case, the lack of improvement is most likely caused by the fact that the mechanism for enabling memoization is a greater overhead than the averted computations themselves. Admittedly, the functions chosen for memoization, from both tested repositories, have multiple arguments, which is not recommended when considering memoization, as indicated by [12]. Pinto et al. make it a precondition for memoizability for methods in Java to have simple or primitive parameters. This is because, parameters determine the size of the key in the memoization table. In Pinto's case, Java `HashTables` should handle this more efficiently, as data types in Java have their own implementations of the hashing function. In our case, the arguments need to be stringified using `JSON.stringify`, in order to be used as keys. This process can add a significant overhead to the memoization process. An observation that I make at this point is that memoizable functions, as Yang et al. define them, are scarce or inexistent in the repositories that I reviewed in the scope of this project. While an anecdotal observation, the "single immutable parameter, primitive return type, without side effects" type of function does not seem like a very common pattern in *"real-life"* projects. An interesting follow-up project to this one might be to perform a study on the frequency of memoizable functions in existing projects.
2. **Perceivable warmup overhead** - Before having implemented the warmup step in the experiment suite, measurements indicated an initial high outlier, for both libraries. This means that libraries require an initialization phase, at which point they will be less time and energy performant than afterwards. In my experiment suite, the two utilities are not reset between each test case execution, which means that the initialization phase happens once in the entire experiment execution of a library. In reality, uses for these libraries might be independent and initialization might happen for every use. This prompts the need for some form of memoization table caching for libraries.

3. `Unknown` **functions are the most memoizable** - Functions that are classified as `pure`, based on independent purity and whether they do not call any functions or they only call other functions classified as `pure`, are scarce and the ones that are found by `grint` are part of non-refactorable files, such as minimized files or bundles. I refactored `unknown` functions exclusively in order to perform the experiments. This indicates that my method of purity detection is incomplete and it is not enough to find a sufficient amount of functions that are memoizable. Of the `unknown` functions, insight into libraries that are used within the analysed project was necessary, in order to assert purity and memoize safely. Incorporating purity analysis for libraries used by a project being linted may increase confidence in purity assertion such that some of the `unknown` functions can be classified as *more likely to be pure*.

4.1.1 Threats to validity

The measurements used in this thesis result from on experiments where each test case was executed for 100 iterations. This number was chosen based on the convenience of shorter running tests and anecdotal observation of the preponderance of outliers. By manually analysing the data, I noticed a mild frequency of outliers, while the rest of the data tended to be close to the minimum. Graphically plotting the distribution of the data revealed decreasing exponential distributions, which confirmed that the data sets contain measurements mainly concentrating around the minimum with decreasing numbers of outliers, as their magnitude increases. At the same time, while calculating the means and medians of the data sets, I also calculated the standard deviations of the data sets, which were small enough for me to consider the 100 iterations enough to produce reasonably consistent data. However, I do not think that this process is robust enough to prove the statistical significance of the data. Some statistical analysis tools may be called for in the future, in order to assert whether 100 iterations appropriate.

Another point of vulnerability is the fact that the experiments do not take into consideration the temperature the experimental machine, nor the ambient temperature. As remarked in other studies [38, 39], higher ambient and internal temperatures can skew the RAPL measurements such that they introduce more outliers and increase variance between test executions that are supposed to be compared against each other. In future iterations of this project, some sort of temperature control and monitoring mechanism would be preferable, to ensure that outlier are not caused by increased setup temperatures and that compared test cases are executed within similar temperature intervals.

Finally, I would have preferred to execute experiments on a larger, more diverse project base. In this project, I took a look at two repositories and refactored a

modest number of functions in these repositories. The results show changes in performance and, while negative, these changes prove that the refactoring process was impactful in both cases. However, the negative nature of the impact is most probably due to the fact that the chosen functions were not particularly memoizable, for reasons I discuss in previous sections. Memoization may have a positive impact, if applied more selectively, to pure functions that are relatively high in complexity, while receiving less complex arguments and returning less complex values. This would reduce the size of the memoization tables and simplify the processing of the table data, which may improve the memoization overhead. As a topic of future work, I would like to search for such functions in a greater number of repositories and analyse the effect of memoization in those cases. If memoization has a positive impact for a certain type of functions, then this may prompt a new *memoizable* coding style, that promotes the construction of memoizable functions.

5 Conclusion

In this project, I investigate the impact of memoization, as an optimization technique, over performance and energy consumption in JavaScript-written applications. As a means to doing this, I construct `grint`, a linter that processes an entire project directory and classifies functions in one of three purity categories: `pure`, `impure` and `unknown`. Using `grint`, I detect pure or potentially pure functions in two open-source JavaScript repositories, `marked` and `trianglify`, which are functions that can be memoized safely. After refactoring these functions using memoization, I compare the performance of the refactored libraries with that of the original ones, in terms of execution time and energy.

Results suggest that memoization can negatively impact the performance of the application, as the memoized versions of the library displayed consistently worse performance than the originals. This suggests that memoization has a significant overhead that must be surpassed in cost by each original function in order for the memoization of the function to yield performance benefits.

While further investigation is needed, in order to determine what makes a function memoizable and devise some rules for memoization such that it yields performance benefits, I have addressed the research questions presented in the introduction of this thesis as follows in the following paragraphs.

What impact can static analysis have on the energy consumption behaviour of a JavaScript application? Static analysis, in the form of purity linting, has proven useful in finding pure functions in a project. The functionality of `grint` is rather conservative with its classification of functions and, thus, most pure functions are catalogued as `unknown`. Even so, this method of static analysis eliminates a great bulk of impure functions, leaving behind worthwhile suggestions for memoizable functions. Refactoring these functions by using memoization did not yield positive results. Nevertheless, the static analysis I perform within this project emphasized portions of code that are safe to optimize by memoization, without changing the functionality of the application. As such, static analysis can reveal code that can be memoized. Its overall impact on energy consumption would depend on some sort of analysis on the functions, that asserts whether they should be memoized. Within this project, results suggest that reckless memoization can definitely worsen the energy consumption rate of an application.

How can we define purity for subroutines in an imperative programming language, such as JavaScript? In this project, I propose a classification of purity analysis for JavaScript functions: independent and contextual. Furthermore, independent purity is comprised of several checks that ensure a function does not

break the established rules of purity (side effects, external dependencies, obligatory return statement, etc.). Contextual purity analyses the functional dependencies of a function and asserts whether any impure operations are performed via the function calls of said function. As such, within the context of this project, I define a JavaScript function as being pure if it respects both independent and contextual purity rules.

What impact does the memoization of pure subroutines have on the energy consumption of an application? After performing a series of test cases on two different open-source repositories, measuring energy consumption for these experiments and comparing the data between memoized and original versions of the applications, results suggest that memoization can have a negative impact on the energy consumption level of an application.

How can we build a static analysis tool that detects and refactors memoizable functions? Finally, this project is facilitated by `grint`, a linter for detecting functional purity. Over the course of this thesis, I explain its architectural details, its practical use case and its shortcomings. At the moment of writing, I meant this thesis as almost a step-by-step "cookbook" to constructing such a tool. While there are certainly points of improvement from which the experimental side of this project would have benefited, `grint` has proven to ease and facilitate the detection of functions that can be memoized safely.

References

[1] N. Jones, "How to stop data centres from gobbling up the world's electricity," in *Nature 561.7722*, 2018, pp. 163–167.

[2] U. of Liverpool, "What is green IT?" https://www.liverpool.ac.uk/sustainability/on-campus/green/green-it/ (accessed Jun. 12, 2021).

[3] "How the earth can benefit from green ICT," 2011. https://www.eurescom.eu/news-and-events/eurescommessage/eurescom-message-archive/eurescom-messge-2-2011/how-the-earth-can-benefit-from-green-ict.html (accessed Jun. 12, 2021).

[4] M. Zimmermann, "Green software: An overlooked factor in the sustainability discourse," 2020. https://www.digitalsme.eu/green-software-an-overlooked-factor-in-the-sustainability-discourse/ (accessed Jun. 12, 2021).

[5] T. Constantin, "An investigation into benchmarking of energy consumption with various programming languages," Jan. 2021.Available: https://projekter.aau.dk/projekter/da/studentthesis/an-investigation-into-benchmarking-of-energy-consumption-with-various-programming-languages(db2be832-c17c-4c6f-a51e-3b31436bf2ce).html

[6] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for android in the large and in the wild," in *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*, 2020, pp. 217–228. doi: 10.1109/SANER48275.2020.9054858.

[7] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th international conference on software engineering (ICSE)*, 2013, pp. 92–101. doi: 10.1109/ICSE.2013.6606555.

[8] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of android apps," Jan. 2018.

[9] R. Morris, "Stephen curtis johnson: Geek of the week," 2009. https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/stephen-curtis-johnson-geek-of-the-week/ (accessed Jun. 12, 2021).

[10] S. C. Johnson, "Lint, a c program checker," in *COMP. SCI. TECH. REP*, 1978, pp. 78–1273.

[11] I. Manotas *et al.*, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 237–248. doi: 10.1145/2884781.2884810.

[12] A. Pinto, M. Couto, and J. Cunha, "Memoization for saving energy in android applications: When and how to do it." Submitted.

[13] "PYPL PopularitY of programming language," 2021. https://pypl.github.io/PYPL.html (accessed Jun. 12, 2021).

[14] "TIOBE index for june 2021," 2021. https://www.tiobe.com/tiobe-index/ (accessed Jun. 12, 2021).

[15] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages," 2017. doi: 10.1145/3125374.3125382.

[16] J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, "Towards purity-guided refactoring in java," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, 2015, pp. 521–525. doi: 10.1109/ICSM.2015.7332506.

[17] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter, "Detecting function purity in JavaScript," Sep. 2015. doi: 10.1109/SCAM.2015.7335406.

[18] "ECMAScript® 2020 language specification," 2020. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/ (accessed Jun. 12, 2021).

[19] "V8 JavaScript engine." https://v8.dev/ (accessed Jun. 12, 2021).

[20] "SpiderMonkey JavaScript/WebAssembly engine." https://spidermonkey.dev/ (accessed Jun. 12, 2021).

[21] "ChakraCore." https://github.com/chakra-core/ChakraCore (accessed Jun. 12, 2021).

[22] "The ESTree spec." https://github.com/estree/estree (accessed Jun. 12, 2021).

[23] "Visitor design pattern." https://www.geeksforgeeks.org/visitor-design-pattern/ (accessed Jun. 12, 2021).

[24] J. Yang, K. Hotta, and S. Higo Yoshikiand Kusumoto, "Revealing purity and side effects on functions for reusing java libraries," in *Software reuse for dynamic systems in the cloud and beyond*, 2014, pp. 314–329.

[25] J. Chege, "Introduction to memorization in JavaScript," 2021. https://www.section.io/engineering-education/an-introduction-to-memoization-in-javascript/ (accessed Jun. 12, 2021).

[26] "Astring." https://github.com/davidbonnet/astring (accessed Jun. 12, 2021).

[27] "JSDoc." https://jsdoc.app/ (accessed Jun. 12, 2021).

[28] "Disabling and enabling inspections." https://www.jetbrains.com/help/webstorm/disabling-and-enabling-inspections.html#suppress-inspections (accessed Jun. 12, 2021).

[29] "Prettier." https://prettier.io/ (accessed Jun. 12, 2021).

[30] "Git - gitignore documentation." https://git-scm.com/docs/gitignore (accessed Jun. 12, 2021).

[31] S. Hsu, "JS objects and arrays — which one is faster?" https://sherryhsu.medium.com/js-objects-and-arrays-which-one-is-faster-cfcdb1281704 (accessed Jun. 12, 2021).

[32] "Phaser - HTML5 game framework." https://github.com/photonstorm/phaser (accessed Jun. 12, 2021).

[33] "Marked." https://github.com/markedjs/marked (accessed Jun. 12, 2021).

[34] "Trianglify." https://github.com/qrohlf/trianglify (accessed Jun. 12, 2021).

[35] "Awesome open source." https://awesomeopensource.com/ (accessed Jun. 12, 2021).

[36] S. Glen, "Sample size in statistics (how to find it): Excel, cochran's formula, general tips." https://www.statisticshowto.com/probability-and-statistics/find-sample-size/ (accessed Jun. 12, 2021).

[37] "Ts-node." https://www.npmjs.com/package/ts-node (accessed Jun. 12, 2021).

[38] K. Khan, M. Hirki, T. Niemi, J. Nurminen, and Z. Ou, "RAPL in action: Experiences in using RAPL for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, Jan. 2018, doi: 10.1145/3177754.

[39] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017, doi: 10.1145/3133876.

[40] S. Pandruvada, "Running average power limit – rapl," 2014. https://01.org/blogs/2014/running-average-power-limit- (accessed Jun. 12, 2021).

[41] "Intel® 64 and IA-32 architectures software developer's manual: Volume 3B: System programming guide, part 2," 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf (accessed Jan. 01, 2021).

[42] "Msr(4) - linux manual page," 2009. https://man7.org/linux/man-pages/man4/msr.4.html (accessed Jun. 12, 2021).

[43] R. Pereira *et al.*, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 2017, pp. 256–267. doi: 10.1145/3136014.3136031.

[44] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages," 2017. doi: 10.1145/3125374.3125382.

[45] MDN, "Tools/power/rapl," 2019. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/tools_power_rapl (accessed Jun. 12, 2021).

[46] "The wilcoxon signed-ranks test calculator." https://www.socscistatistics.com/tests/signedranks/ (accessed Jun. 12, 2021).

Appendix

Time results for both repositories in milliseconds

Library	Workload	Type	Average time	Median time
trianglify	1	memo	38.9162	36.0814
trianglify	2	memo	241.9750	197.0238
trianglify	3	memo	811.8182	691.7651
trianglify	4	memo	2312.6016	2275.8071
trianglify	5	memo	5989.4759	4749.6002
trianglify	1	norm	9.6346	8.6823
trianglify	2	norm	33.1952	26.7609
trianglify	3	norm	70.1137	57.7227
trianglify	4	norm	140.4359	122.1158
trianglify	5	norm	237.7437	176.9045
marked	1	memo	1.7989	1.5343
marked	2	memo	13.2919	12.0194
marked	3	memo	47.7276	42.8440
marked	4	memo	198.3689	184.0369
marked	5	memo	201.6011	188.3333
marked	1	norm	1.6262	1.3084
marked	2	norm	10.7715	10.0003
marked	3	norm	40.2130	36.4140
marked	4	norm	172.1259	164.1545
marked	5	norm	178.1635	169.3855