Refactoring Functional-Style JavaScript Code: Implications on Performance and Readability

Rohit Gowda(2404248)
Masters Distributed Software Systems
Program Testing and Analysis,TU Darmstadt

Subhadeep Manna(2793470)
Masters Distributed Software Systems
Program Testing and Analysis, TU Darmstadt

Abstract

JavaScript contains many features of functional programming, mostly some popular native functions such as ForEach, reduce, map and filter. Functional programming style is usually less complicated to understand and work with than imperative style however its a common assumption that it is less efficient. This could lead to lower performance of the code. In this project we plan to estimate the performance gains which can be achieved by refactoring the JavsScript code to iterative style and how it might effect the readability/maintainibiliy of the code. We also try to provide our understanding where the performance gain would be desirable over readability and vice-versa.

Keywords Abstract Syntax Trees, Traversal, Performance, Cyclomatic Complexity(esc), Readability, Lines of Code.

1. Introduction

The fundamental differences between functional and iterative styles of programming would usually account for the differences in performance and readability of the code. We would briefly summarize each of style for better understanding before we discuss our approaches for refactoring.

Functional programming could be described as a form of declarative programming. A functional style programming is the process of building software by composing pure functions. It primary motive is to use transformations needed for the informations desired. Thus, it accounts for high code readability or maintainability. However, could give rise to lack of performance(Selakovic and Pradel 2015). An imperative style programming is where the programmer tries to specifically instructs the computer about the step it must take to accomplish the goal. Tracking the changes in state as opposed to its functional counterpart thus being step by step algorithmic style.

The benefit of adapting functional over imperative could also account to lowering the cyclomatic complexity of the code of the code. Code metrics which would be essential in picking one style over the other is discussed in code metrics comparison section of this document.

Below are the simple examples of functional and imperative style coding:

Functional Style from package rimraf v2.5.1 (rim)

```
var methods = ['unlink','chmod', 'stat', 'lstat
    ', 'rmdir', 'readdir'];
methods.forEach(function(m)
{
    options[m] = options[m] || fs[m]
    m = m + 'Sync'
    options[m] = options[m] || fs[m]
})
```

Equivalent Iterative Style

```
var arrayUsingItr = methods;
var newArray = [];
for (var i = 0, counter = 0; i < arrayUsingItr.
    length; i++) {
        var m = arrayUsingItr[i];
        options[m] = options[m] || fs[m];
        m = m + 'Sync';
        options[m] = options[m] || fs[m];
        newArray[counter] = arrayUsingItr[i];
        counter++;
    }
methods = newArray;
```

In the two code snippets above, the functional style is much shorter and more understandable. However,it raises question on the difference between the performance and its effective implementation of these styles according to situations considering their other metrics such as readability, maintainability and Lines of Code.

Figure 1. Iterative vs Functional Performance.

The figure above shows the difference in performance when using the two separate styles. It is tested based on an array of 1500 elements using native javascript map function and an equivalent imperative style.

So, in this project we implement a refactoring tool which helps to convert most common functional paradigms such as ForEach, Map, Reduce and Filter into their equivalent iterative style. We have used five libraries for testing of our tool(semver (sem), rimraf(rim), minimist(min), nanomist(nan) and loadash). Upon successful refactoring we perform a detailed analysis on the performance gain or loss over the original code.

We discuss our approach on the refactoring mechanism in section 2. We focus on the basics of Abstract Syntax Trees and their manipulation which are used for refactoring. In the following section, we take a closer look on how

our implementation of this approach works. We briefly discuss different parts of the refactoring tool and describe the functionality of each part. The next part of this document is dedicated to results of refactoring and the approaches for validating them. Consequently, we do a performance analysis on the original and refactored code. We compare the results for each of the libraries individually. The penultimate section covers the code metrics and their estimations of the refactored code. Thus, the performance and code metrics provides us with essential data which helps in the comparison of the functional approach and the imperative approach over each other.

2. Approach

JavaScript has to be converted to a suitable form before it can be manipulated or the programming constructs can be analysed. Generating a Abstract Syntax Tree(AST) from a given javascript code serves this purpose. An AST is an essential syntactic representation of the source code. Each node of the tree represents a programming construct in the source code. Thus, this is our first step in our approach for building the refactoring tool. Once the tree has been constructed using a suitable parsing tool such as Esprima or Acorn we traverse the tree. Traversing the tree enables to recognize Functional patterns which has to be converted to iterative form. Once such a pattern is detected we need to manipulate the AST and replace the Functional types into iterative types. Tree traversal libraries such as Estraverse helps us accomplish this step. The approach in brief is as follows:

1. Static Aanalysis of AST
2. AST Manipulation
3. Code Generation

2.1 Static Analysis of AST

Static analysis is a way of analysing the code without actually running the code. In this section we generate an Abstract syntax tree(AST) from a static code using JavaScript parsing tools Esprima. Esprima is a library to perform lexical and syntactical analysis on the JavaScript code it provides JSON formatted AST as an output. Since Esprima itself is written in JavaScript it runs on various JavaScript environments.

While AST parsing using Esprima for the original code of the given libraries(minimist, optimist, semver, rimraf, lodash) we do not actually run these libraries. Hence making it a static analysis. Below is an example of AST parsing using Esprima for a map function which converted into an abstract syntax tree.

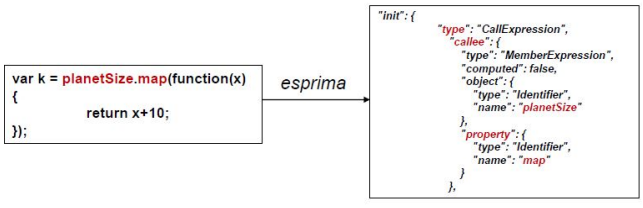

Figure 2. AST parsing of map function

2.2 AST Manipulation

Once the AST is generate we need to traverse the syntax tree. While we could write a custom tree traversal algorithm we used Estraverse (est) instead for AST traversal. Estraverse comes with its own set of traversal functions which proved to be real handy. Some of the functionality which we used are as follows:

1. **Enter**: Enter node and traverse child nodes recursively.
2. **Replace()**: Replace a node with another node.
3. **Skip()**: Skip the traversal of child nodes.
4. **Break()**: Discontinue the current node traversal.

The patterns for native functional styles like ForEach, map, reduce and filter are searched and collected from the AST based on the scope of each element. After a pattern is detected a node transition which would result in iterative form is carried out. It is important to maintain the position and value of the parent node(s) while removing or replacing nodes.

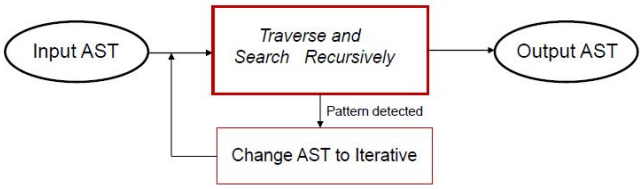

Figure 3. Tool Approach

3. Refactoring Tool Approach

Based on the approach which we described in the previous section we implemented the tool having three essential parts. They are as follows:

1. Pattern Detection Engine
2. Data Store
3. Template Creation and AST Insertion

3.1 Pattern Detection Engine

Pattern detection for the given functional types is based on the parent child relationships in the structure of the generated AST. For Example, a pattern for map would have a callee, property, argument and body elements. For a successful detection of every map its is important to identify the functional pattern. A relationship could be established by taking a look at the structure of the AST. This is useful for providing scoping of the node and other tokens.

```
if(node.callee)
        {
          if(node.callee.property)
            {
              typeOfOp=node.callee.property.
                  name;
            }
          }

          if(typeOfOp==<operatorName>) {return
              obj;}
```

Based on Figure 2 simple map pattern. Our detection engine snippet showed in above figure would be able to detect it based on attributes of a Node.type: CallExpression, Node.callee.object.name:planetSize and Node.property.name:map. Here the dot operator "." is used to separate each attribute from parent to child which is from left to right respectively.

The AST is traversed recursively for pattern detection. The value of type of operator(typeOfOp) is matched with the given functional names to lock a definite pattern.

3.2 Data Store

Once a pattern has been detected by the pattern detection engine. We store the detected node values according to the scope encountered. The unique identifier where the pattern belongs is given by two keys. First is the parent of the node where the pattern got detected and second is the node itself. The data store is of format simple JSON which stores the values that could be substituted into a pre-defined template having a generic iterative format. The following format is used for four functional patterns:

```
{"parentSignature":parentSign,"bodySection":
    bodySection, "returnCondition":
    returnCondition, "arrayOperatedOn":
    arrayOperatedOn, "argVariableName":
    argVariableName};
```

The key value pairs used in the Data Store is unique and hence significant in identifying each of the instances while substitution. "ParentSignature" captures the parent-node of the node where the functional style was detected. "BodySection" stores the body of the functional style node which was detected. Return Condition could be used to substitute into a conditional construct while creation of filter, map and reduce from a template. Array or collection operated upon is stored in "arrayOperatedOn". The argument of the function used as a callback is stored in "argVariableName".

3.3 Template Creation and AST Insertion

The values from the data store are used to create partially iterative ASTs from a predefined template which are different for each of cases(forEach, filter, Map and Reduce).

```
//1. Name of the array on the Right
var arrayUsingItr=null;
//2. New array for storing values
var newArray=[];
for(var i=0,counter=0;i<arrayUsingItr.length;i
    ++)
{
  //3.functional args replaces val
  var val=arrayUsingItr[i];
  //4.Put the return condition here
  if(true)
  {
    newArray[counter]=arrayUsingItr[i];
    counter++;
  }
}
```

The code snippet above shows a predefined template in iterative form. The suitable values are substituted into appropriate fields of these templates. Insertion of the partial ASTs into the main program AST would create iterative counterparts of the functional ASTs. The newly created partial ASTs replaces the nodes marked by insertion points in the ASTs. This is based on the unique combination of node and parent signature. Once such a position is detected the partial ASTs from the store are inserted serially by popping the data store. Thus removing the chances for mismatch or incorrect replacement.

3.4 Code Generation

This is the third step of our approach, here the code is generated for manipulated AST. This is the final step to get refactored iterative style of code out of original functional style using the library Escodegen(escodgen).

```
Example: RefactoredCode = escodegen.generate(
    Changed AST)
```

Escodegen is an ECMAScript code generator from the Mozilla Parser API AST.

Helper and utility functions are used to clean up the code after the functional AST is converted to iterative AST. The clean up usually removes unused or redundantly generated variables or constructs. After this step is completed the refactored code for all the libraries should be checked against all the original code for the percentage of the success rate.

4. Results and validation

To check the correctness of the code refactorings we used the implicit testSuite provided by each of the following libraries. The command **"npm test"** starts the testing of the converted code.

The validity was considered based upon the tests passed. Suppose, for the refactored code a single test failed then that was replaced with the original code. We listed our findings in the tables below for each of the following libraries for which we tested. The legend for the table is as follows **(FST:Functional Style,TI:Total Instances, DT: Detections, SC: Successful Conversions, UC: Unsucessful Conversions)**

FST	TI	DT	SC	UC
ForEach	3	3	2	1
Map	10	10	9	1
Reduce	0	0	0	0
Filter	2	2	1	1
False Positives	15-15 = 0			
Success Rate	12/15 = 0.80			

Table 1. Semver.js

FST	TI	DT	SC	UC
ForEach	4	4	3	1
Map	0	0	0	0
Reduce	0	0	0	0
Filter	0	0	0	0
False Positives	4-4 = 0			
Success Rate	3/4 = 0.75			

Table 2. Rimraf.js

FST	TI	DT	SC	UC
ForEach	12	12	7	5
Map	0	0	0	0
Reduce	0	0	0	0
Filter	3	3	0	3
False Positives	15-15 = 0			
Success Rate	7/15 = 0.46			

Table 3. Minimist.js

From the validity of the results it can be seen that the library semver has the most successful conversion rate. This is due to the reason that most of the libraries uses complex functional patterns which chains two or more function together. Although, the tool supports refactoring of nested functions of same type chained function and nested function of different type are still a challenge for the refactoring tool.

FST	TI	DT	SC	UC
ForEach	12	12	7	5
Map	0	0	0	0
Reduce	0	0	0	0
Filter	3	3	0	3
False Positives	15-15 = 0			
Success Rate	7/15 = 0.46			

Table 4. Nanomist.js

4.1 Performance Testing Methodology

We performed the performance testing by creating a "childProcess" using nodeJS npm package "child-process". This enabled us to spawn a child which could run multiple testSuite intrinsic to the refactored library certain given number of times. We used the library "performance-now" to benchmark timings in this process.

We made sure that the child process forked was synchronous so the testSuites were executed serially. We took samples for given number of iterations(10 iterations) of the original code vs the refactored code. Thereby, comparing them.

```
var exec = require('child_process').execSync;
var now=require('performance-now');
var cmd = "npm test";

var options = {
  encoding: 'utf8'
};

//Test Suite for 1 iterations
...
//Test Suite for 10 iterations
....
//Test Suite for 20 iterations
var t0 = now();
for(var i=0;i<20;i++)
{
        exec(cmd, options);
}
var t1 = now();
console.log("Time taken for 20 Test iterations
    ", (t1-t0).toFixed(3),"milliseconds");
```

We then took two more samples for performance(1 iterations and 20 iterations) so that we may compare if they are statistically significant. The results for these are present in spreadsheet externally. We used the npm package "ttest" (tte) to detect whether two samples are statistically significant of each other.

4.2 Performance Analysis

Performance is given by verifying individual test suits against Refactored code and Original code for each of the test libraries and is presented in form of graph for better understanding. Lower time captured by the testSuite means better the performance of the code.

Performance here is calculated by the time taken to run iterations of testSuite over each libraries. Here we have considered ten iterations.

The chart presented in the next figure compares the time output by the testSuite over the original and refactored versions of semver.js libary. The time is measured in seconds. Although, the filter has a single instance which has been refactored it does not perform better as compared to the original version. ForEach instances which were refactored were much higher compared to the filter instances thus we can see a reduction in time for testSuite executions. Map instances also gets a fraction of seconds faster compared to filter. Again, this is the case where number of instances converted are much higher than the single instance of filter.

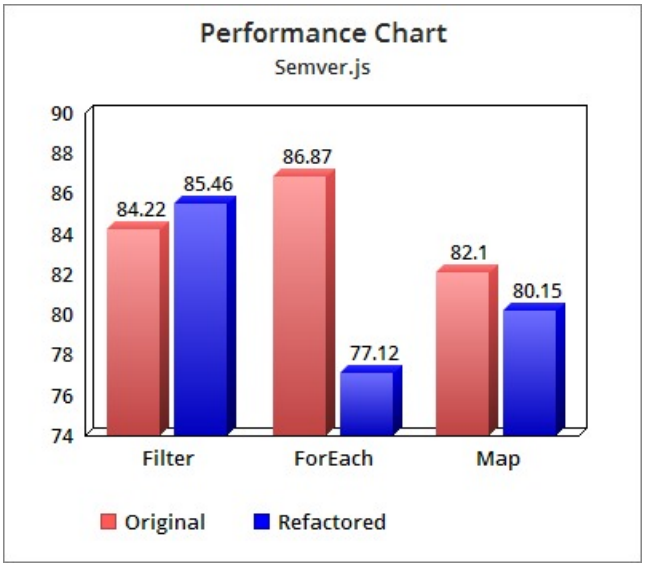

Figure 4. semver

The chart below shows the differences in time between rimraf library refactorings and the original. ForEach instances had a very little increase in performance when three out of four instances were refactored.

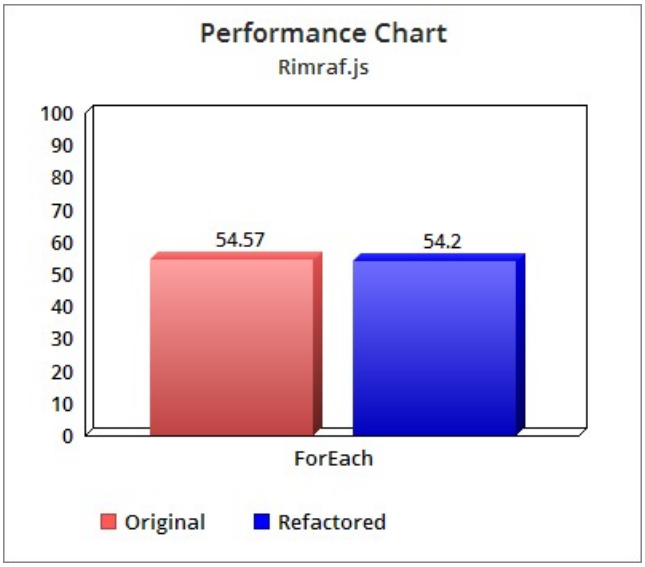

Figure 5. rimraf

The chart below shows no improvement on the performance after forEach refactorings on minimist package.

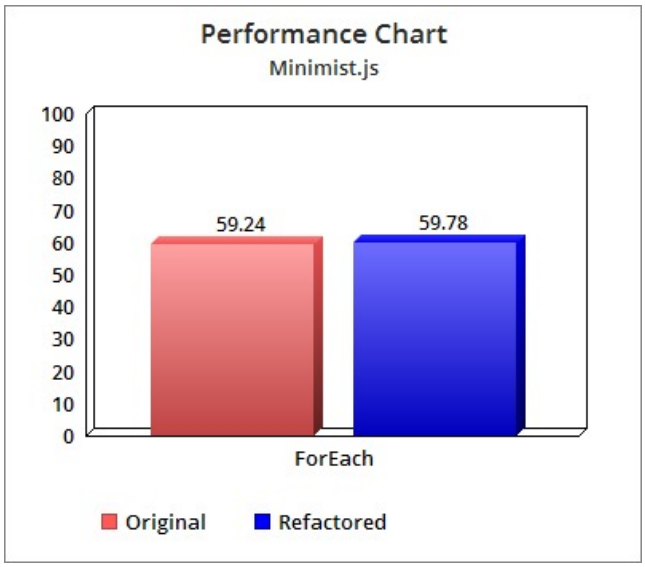

Figure 6. minimist

The chart below shows a slight improvement on the performance after forEach refactorings on nanomist package. Which is a similar package to minimist.

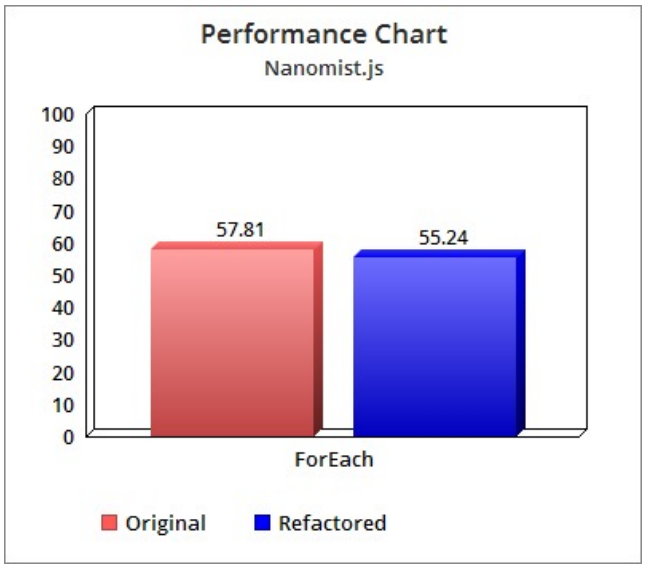

Figure 7. nanomist

4.3 Statistically Significant Refactorings

While the performance test data for single values(10 iterations) could produce a random behaviour. The distribution over higher number of test sets could help us determine whether the refactorings per library i.e, original vs refactored is statistically significant or not. The table below provides the results for the ForEach 2 sample test containing(1 iteration, 10 iteration and 20 iteration)time calculated for iteration is in seconds. The table shows data only for refactored code:

LIB	1 Itr	10 Itr	20 Itr	Alpha	valid()
semver	8.25	78.71	147.44	1	True
rimraf	9.42	66.16	115.11	1	True
minimist	6.10	53.21	114.17	1	True
nanomist	7.12	58.41	115.47	1	True

Table 5. ttest Statistics

The above data implies that the refactoring conducted for ForEach on each of the libraries is statistically significant and provides an increase in performance over the original code. The full data is available in the spreadsheet in the repository.

5. Code Metrics

We consider the Halstead software metrics to analyse the code metrics. We consider metrics like Cyclomatic Complexity, Readability/Maintainability and Lines of Code(LOC).

Given below is brief summary of each of the metric we used for code analysis:

Cyclomatic Complexity This is a software metric used to indicate the complexity of the program, It is a quantitative number of distinct(independent) path taken in the code. Lower the complexity is better. If a source code does not contains any control flow statements such as conditions and decision statements the complexity would be one. Since our refactored code contains iterative function, its complexity will be more compared to the original functional style program.

Maintainability Index This gives a quantitative measurement on relative ease on maintaining a code. This is often termed as Readability of the code. Higher maintainability index results in ease of code understanding.

Refactoring the code to iterative style results in lower maintainability index. This is because it would take more effort to scan and understand the code.

LOC Lines of Code is the number of logical lines of source code. Refactoring to iterative style increases the amount of lines of code. Which could cause code lint and redundancy.

The code metrics were estimated using escomplex and visualized using Plato javascript packages.

The Code Metrics Report of Semver.js, Rimraf.js, Minimist.js, Nanomist.js are present in results folder in the repository. The table below compares ForEach cyclomatic complexity(CC), Maintainibility index(MI) and LOC:

LIB	CC	MI	LOC
semverOrig	235	64.21	1296
semverRef	237	63.62	1314
rimrafOrig	74	65.42	335
rimrafRef	77	62.42	353
minimistOrig	70	61.59	235
minimistRef	76	54.11	286
nanomistOrig	70	61.59	235
nanomistRef	77	53.18	274

Table 6. Code Metrics

From it table above it is evident that refactored code results in lower maintainability index, low cyclomatic complexity and more lines of code. This is bad indicator regarding the readability and understanding of the code. However, there is a gain in performance whenever the code metrics show a negative indicators on a refactored code.

6. Conclusion

In this document we tried to answer the question *when* and *how* the gain in performance is significant enough to consider refactoring of functional to iterative style. From our findings we would like to imply the following points:

1. Performance gain is higher for higher number of functional statements refactored.
2. Performance gain often comes at a cost of poor maintainability and higher cyclomatic complexity of the code.
3. Performance also depends on type of data used for test cases.
4. Performance gain also depends on type of test cases.

Thus we should consider refactoring for performance gain only if there is enough gain to balance the maintainability and higher cyclomatic complexity of the code. Even, minor refactorings often result in changes in the code metrics. So if the changes in code metrics are large enough for minor performance gain then we should revert to the original code.

Bibliography

Software complexity analysis of javascript-family abstract syntax trees. `https://github.com/philbooth/escomplex`. Accessed: 2018-02-05.

Ecmascript js ast traversal functions. `https://github.com/estools/estraverse`. Accessed: 2018-02-05.

Test library. `https://github.com/substack/minimist`. Accessed: 2018-02-05.

Test library. `https://github.com/zeit/nanomist`. Accessed: 2018-02-05.

Test library. `https://github.com/isaacs/rimraf`. Accessed: 2018-02-05.

Test library. `https://github.com/npm/node-semver`. Accessed: 2018-02-05.

Perform the student t hypothesis test. `https://www.npmjs.com/package/ttest`. Accessed: 2018-02-05.

M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. 2015.