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

Course Project proposed by Marija Selakovic
Program Testing and Analysis – Winter 2017/2018

Introduction

JavaScript incorporates features of functional programming, mostly with its native functions, such as *forEach*, *map*, *reduce* and *filter*, and with libraries such as *Underscore.js*. An important benefit of adapting a functional over imperative style is better readability and maintainability of the code.

To illustrate benefits and drawbacks of functional programming in JavaScript, consider the following problem: concatenating all sub-arrays of a given array. In JavaScript this can be done imperatively by using nested *for* loops as follows:

```
var flattened=[];
for (var i=0; i<input.length; ++i) {
    var current = input[i];
    for (var j=0; j<current.length; ++j)
        flattened.push(current[j]);
}
```

However, the same functionality can be accomplished with the built-in *reduce* method:

```
var flattened = input.reduce(function(a, b) {
    return a.concat(b);
}, []);
```

The second solution is much shorter and easier to understand (for those familiar with *reduce* method). The biggest benefit of functional over imperative style comes when chaining several functional style methods, which leads to a very readable and understandable code.

However, functional style is usually less efficient. Inefficient iteration due to functional programming is reported as a common root cause of performance issues in JavaScript code [1]. Considering the second solution, for each element of the input, a new array is created - the concatenation of a and b. On the contrary, the first solution just uses a single array.

The general belief is that imperative loops perform faster than functional-style methods. However, the interesting question is when and how often the performance difference gets significant? Furthermore, it is unclear in which situations the functional style provides bigger benefit on readability and maintainability? For example, sometimes an imperative loop solution is not that complex but performs better.

Goal

The goal of this project is to explore the implications on performance and readability/maintainability of refactoring functional-style JavaScript code. The first step is to develop a refactoring tool based on AST transformations for the most common functional-to-iterative coding patterns. These patterns should include rewriting rules for *map*, *filter*, *forEach* and *reduce* methods.

The second step includes an implementation of an analysis that measures code complexity based on common metrics: cyclomatic complexity and/or number of statements. The final step of the project consists of evaluating changes in code complexity and performance of refactored programs. The report should contain a description of coding patterns, rewriting rules, programs considered in the evaluation, refactorings performed for each program and implications on program's readability and performance.

Tasks

More specifically, the project involves the following tasks:

- Get familiar with functional style methods in JavaScript and define most common functional-to-iterative refactoring patterns.
- Get familiar with libraries for manipulating ASTs, specifically esprima[1], estraverse[2] and escodegen[3].
- Design and implement a refactoring tool based on AST transformations for defined patterns.
- Implement or reuse some of the existing tools[4] that measures cyclomatic complexity and number of statements in JavaScript program.
- Run the refactoring tool on the following libraries: *minimist*[5],*optimist*[6],*semver*[7],*rimraf*[8] and *lodash*[9].
- For each refactoring, run the modified version of a program using a test suite available in the program repository. If all tests pass, collect the measurements on complexity metrics. Finally, run both, the original and refactored program with a test suite and statistically evaluate the performance differences between these two versions.

Deliverables and Grading

The team must present the project between February 12 and 16, 2018, in a short talk, followed by a question and answer session. The exact time and date will be agreed on in due time. The team must submit the final project report and the project's implementation on February 25, 2018. The report and the implementation must be send to the project mentor via email (or, for large files, a file hosting service, e.g., Dropbox).

Grading will be based on the following criteria:

Criterion	Contribution
Report (structure, explanations, examples, writing)	20%
Approach (own ideas, independence, involvement, organization)	20%
Results (discussion and interpretation, soundness, reproducibility)	20%
Implementation (completeness, documentation, extensibility)	20%
Presentation (clarity, illustration, quality of answers)	20%

Reading material

[1] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72, New York, NY, USA, 2016. ACM.

[1] https://github.com/jquery/esprima
[2] https://github.com/estools/estraverse
[3] https://github.com/estools/escodegen
[4] `https://github.com/escomplex/escomplex`
[5] https://github.com/substack/minimist
[6] https://github.com/substack/node-optimist
[7] https://github.com/npm/node-semver
[8] https://github.com/isaacs/rimraf
[9] https://github.com/lodash/lodash