Paper Title*

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

***Abstract*—JavaScript is the most dominant language used to develop front-ends and back-end in multiple web applications. This is because of multiple features it brings with it along the lines of event-driven and asynchronous programming; which are one of the most crucial aspects when developing a smooth and fast web interface. One of the features that helps maintain flow of instructions in asynchronous language is callbacks or higher order functions. However, a poor understanding of callbacks can result in many complexities such as nested callbacks (also referred to as callback hell). Another problem that nested callbacks can cause is lose of benefit of asynchrony since execution of program becomes linear when independent functions are nested into callbacks of other function calls. We present a tool developed in JavaScript that checks for independent instructions inside callbacks and refactors them outside of callbacks.**

***Index Terms*—JavaScript, Callbacks, Refactoring, Asynchronous programming**

I. Introduction

Callbacks if implemented rightly can result in a very smooth and robust functioning of program, however, callbacks in JavaScript are inherently difficult to understand. They get even more complicated when there are callbacks nested within other callbacks. In addition to increasing complexity in understanding code, they can also result in loss of benefits that JavaScript brings on table with its asynchronous way of executing instructions. Consider the code excerpt in figure 1.1, we see in line number 14 a call to readfile for file1.txt with a callback, in its callback there is another call to same function with file2.txt. Execution of this code will result in a completely serial reading of files, where as we could have achieved better results if second call was outside the callback of first call. On average, every tenth function in JavaScript user applications takes a callback as an argument, over 43% of these callback functions are anonymous and majority of these callbacks are nested[CITATION Gal15 ł1033]. This naturally means, there will be many such situations where callbacks will be affecting asynchrony and their refactoring will not result in any change in execution of program. This gives us opportunity to develop a method that automates refactoring in JavaScript function callbacks in such a way that independent instructions are taken out of callbacks and placed in a scope where their earlier execution does not affect functionality of program but enhances the overall performance.

Rest of the articles is divided into following order:

- Methodology
- Implementation
- Limitations and Challenges
- Related Work
- Conclusion

II. Methodology

Our proposed solution refactors all the independent instructions from callbacks, these can be reduced and limited to only async I/O or other blocking instruction but for simplicity we will currently refactor all independent instructions. Our solution mainly consists of 3 major parts:

- AST extraction
- Execution context extraction
- Decision based refactoring

A. AST extraction

Abstract Syntax Tree is the context form of a parse tree that contains information about order of different instructions, operators and variables. An AST assigns a separate node to every operator or operand it encounters in code; thus, making a full fledge tree for the whole syntax of program. There are multiple different tools available that produce detailed AST for any given code. We can extract information like names of variables, scope of variables, structure of functions and metadata like line number in actual code etc.

B. Execution Context extraction

Next, we use AST to develop execution context of our program. Execution context is the environment, different functions execute in. Typically, execution context contains information such as names of variables, scopes of variables, variables

declared within a function and global variables. To extract execution context, we maintain dictionaries for different variables: for variables in scope of each function we maintain a separate dictionary, for variables declared in a function we have a separate dictionary and for arguments of function we have a different dictionary. We traverse the whole AST recursively from root to leaves and at each node we check if its a variable we append it to appropriate functions list. At the end of this function we have complete execution context of the program. Pseudo code for Execution Context Extraction is shown in figure 2.

C. Decision based refactoring

After we have execution context and we know scopes of different variables, we can now make decisions about what instructions are refactorable. We can refactor only a few types of instruction, the fact that what instructions are fit for refactoring depends upon what variables they depend upon. In this perspective we can have following 4 cases:

- **Instruction depends upon no variable:** There can be instructions that do not depend upon any of the variables for example declaration of a new variable, or line number 15 from figure 1 is fit to be refactored. In this case, instruction is safe to be refactored.
- **Instruction depends upon function argument:**Since callbacks may include instructions that are dependent upon arguments of main function, we should not refactor these instructions out of function. Figure 3 shows another version of code from figure 1 where line 2 is not refactorable now.
- **Instructions depend upon variables declared in scope of parent function:** In this case, such an instruction will be safe to be refactored. Consult figure 4, where line number 2 is refactorable now because it has been declared in scope outside of containing function.
- **Instruction depends on a variable declared within the scope of function:** This case can have two scenarios, either the variable in terms of which the instruction is expressed is itself refactorable or it is not. In first case, when variable itself is refactorable, we can first let variable declaration instruction get refactored first and then refactor the instruction that contains the said variable. In other case when variable cannot be refactored, we do not refactor any instruction that contains it.

Now that we have mentioned decisions that will be governing decision of refactoring, we can move to actual methodology of refactoring. Refactoring function works recursively in bottom up manner, refactoring the deepest of callbacks first and then propagating refactoring towards the containing function. Figure 5 shows pseudo code for refactoring. For each node we recursively call refactor for every child and once all children nodes have been refactored we check for every child node with respect to fore-mentioned conditions if child node is refactorable. If it is refactorable, we delete that node from current tree node and add it to parent tree node.

III. Implementation

We have implemented above mentioned program in JavaScript ECMAScript 5 due to some constraints. We use UglifyJS to extract AST since it gives a very detailed tree with metadata such as line number and character number included within the tree. Additionally, it allows easy refactoring and is convertible to code even after we have made changes to tree. Moreover, UglifyJS also provide features such as tree compression and mangling variable names which can be helpful. Since UglifyJS is only compatible with JavaScript ECMAscript 5, we have also implemented our code in same version.

IV. Limitations and Challenges

V. Related Works

VI. Conclusion

Callbacks can ensure very smooth and prompt execution of program if they are implemented rightly. However, since callbacks tend to get complex as they get nested they might even affect asynchrony of program execution. Our approach tends to find such abnormalities in JavaScript programs and improves performance by refactoring such instructions to position where they are executed earlier without affecting functionality of program. We achieve this goal by extracting AST of code using a JavaScript library UglifyJS, next we extract execution context from the AST and finally based on this AST and few rules we choose which instructions can be refactored. These instructions are removed from current functions body and are placed in callee functions body. Our approach currently only works on JavaScript ECMAscript 5 due to compatibility issues of UglifyJS, but it can be extended to newer versions if replacement for UglifyJS is found.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, “Title of paper if known,” unpublished.

[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.