JavaScript Refactoring Tool to Analyze the Usage of 'let' and 'var' Keywords

Sam Lichlyter, Spencer Kresge, Nirvik Das
School of EECS, Oregon State University
Corvallis, OR, USA
{lichlyts,kresges,dasn}@oregonstate.edu

Keywords

JavaScript, Refactoring, Visual Studio Code

1. RESEARCH QUESTIONS

With the *let* and *const* keywords providing a different set of rules in terms of scope, older JS code can be refactored to replace *var*, in places where *let* and *const* would be more appropriate in context. Without an automated tool, such refactorings can be very expensive and tedious, especially in programs with a large number of SLOC. While some of this refactoring could be done using a find/replace method, blindly applying these changes could produce hazardous and unwanted results.

In this paper, we attempt to deal with this issue by answering the following research questions:

RQ1 To what extent has *var* been refactored to *let* and *const* in older applications?

RQ2 What are the challenges that developers face when it comes to implementing such a refactoring procedure manually?

It's important to answer the above questions because the margin of error in such an implementation can be quite large. Human errors and other factors can cause unwanted bugs to be introduced, *var* and *let* act differently on different scopes. Knowing how the scopes affect the source code is quite important to make a successful refactoring implementation.

To answer the above questions, we will develop a tool that can analyze variables and the scope within which each the variable's current value is effective. After confirming the scope within which the variables should be active, the tool refactors *var*, if necessary.

2. TECHNICAL CHALLENGES

In order to understand the usage of *var*, *let*, and *const*, a thorough understanding of scope and assignment is needed. Traditionally developers used *var* as a catch all for variables of all nature. ES6 introduces new and refined constructs for creating variables - *let* and *const*. The keyword *const* is a stronger declaration for variables that never need reassignment. Developers often misuse this new feature by declaring a *var* and then never reassigning it. The keyword *let* is a stronger declaration of scope for variables only used in a particular block.

The technical challenges associated with understanding the usage of these new constructs requires being able to understand the intent of existing code. By acknowledging this intent we can begin to analyze where variable keywords (*var*, *let*, *const*) are being utilized appropriately or not utilized at all. For instance, scanning a file and realizing a *var* declaration is never reassigned would be a great opportunity to use *const*. Realizing that an old *var* declaration is used only in the scope of a single block means it could be more readable using *let*. These opportunities for refactoring are only discovered by scanning the source code for variable references and understanding the semantics.

In order to create a good refactoring tool, we need to understand how developers currently refactor old code into this new paradigm. This hurdle can be bridged by examining a corpus of GitHub commits surrounding lines of code containing the new ES6 keywords. This will not be an easy task since a diverse collection of repositories will be needed - all of which must demonstrate relevant commits. This same corpus can be used to identify challenges in refactoring by tallying the resultant incorrect uses of the new keywords. Incorrect uses will be defined as uses of the new keywords that invalidate their intended use or otherwise cause program errors.

Hopefully, our tool will provide a methodology for solving these technical challenges, answering our research questions, and ultimately improving the developer usage of the new ES6 keywords *let* and *const*.

3. PROPOSED SOLUTION

Our proposed solution will consist of an extension built for the open-source text editor, Visual Studio Code. The tool will be able to be called using VSCode's "Command Palette" as well as run in the background as users code so that if they use a *var* keyword when they should be using *let* or *const*, they will be alerted by a tooltip or a specialized styling of the declaration.

The tool will need to analyze what variables are declared where and check to see if any of them were changed after their declaration and in what scope they are changed. This cannot be done using a simple text analysis of the code, a more powerful static analysis tool will need to be utilized or developed.

DOI: 10.1145/1235

We will first start out by gathering a corpus and answering the aforementioned research questions. This will help us to design a better, more usable tool. We will then build our tool borrowing some techniques from the authors below.

The tool will analyze the usage of variables defined within the project by their respective scopes and determine whether they should be declared using the *var*, *let*, or *const* keyword.

4. RELATED WORK

In this section we'll look at five previous works whose authors ventured into the refactoring of JavaScript code. We'll look at what they did and how it will impact our research.

4.1 Automated Refactoring of Client-Side JavaScript Code to ES6 Modules

This paper looked at taking JavaScript code that was written using previous versions of Javascript, notably ES5 and below and transforming it into ES6 code (the latest version). It mainly focuses on refactoring client-side web applications to fix code smells, specifically those related to global variables and global functions that are declared in JS files linked to web pages. The authors developed an approach to refactor this code and properly scope variables and functions to try and reach 100% encapsulation [5].

This paper provided us with a starting point in how we think we might approach the problem. The method and tools used to provide their analysis could prove useful to us in the development of our new tool.

4.2 Type Refinement for Static Analysis of JavaScript

This paper proposed a new technique for static analysis called *type refinement*. Using their new technique they showed they could improve static analysis precision by up to 86% without degrading performance. This was done by developing a new static analysis tool that would report potential type-errors in JavaScript code [3].

Similar to the paper before it, we can glean from this paper a set of techniques and tools used for static analysis.

4.3 JSNose: Detecting JavaScript Code Smells

The authors of this paper discuss different code smells that are prevalent in JavaScript code. They proposed a set of 13 code smells and developed a detection technique called JSNose. They analyzed 11 web applications to see which code smells were most prevalent. Their findings indicated that lazy objects, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables were the most prevalent code smells [1].

Some of the proposed code smells could be early indicators of where developers may be misusing the *var* keyword, especially the global variable code smell. Again, some of the techniques and tools used in the paper might prove useful in the development of our new tool.

4.4 Tool-supported Refactoring for JavaScript

This is an older paper when tools for JavaScript refactoring were in their infancy. This paper took on the task of analyzing the correctness of refactorings using a technique called *pointer analysis*. This allowed developers to ensure the correctness of their refactorings by having a set of preconditions in which the code had to fulfill in order to be refactored without losing or changing the semantics of the code [2].

While potentially not as influential to us as new tool developers, this paper helped lay the ground-work for modern refactoring and still provides useful information in the area of static analysis.

4.5 Semi-Automatic Rename Refactoring for JavaScript

This paper focused primarily on rename-refactoring. The authors used a technique to find related identifier tokens which allowed them to do their static analysis. In their tests they saw a reduction in manual effort by 57% over search-and-replace methods [4].

This paper provided useful techniques in finding related identifier tokens which we might be able to apply in our new tool to find related identifiers for variables instead of function names.

5. REFERENCES

[1] A. M. Fard and A. Mesbah. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, Sept 2013.

[2] A. Feldthaus, T. D. Millstein, A. Moller, M. Schafer, and F. Tip. Tool-supported refactoring for javascript. In *OOPSLA*, 2011.

[3] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 17–26, New York, NY, USA, 2013. ACM.

[4] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 17–26, New York, NY, USA, 2013. ACM.

[5] A. Paltoglou, V. E. Zafeiris, E. A. Giakoumakis, and N. A. Diamantidis. Automated refactoring of client-side javascript code to es6 modules. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 402–412, March 2018.