Project Report: 14 - Hoisting Nested Functions

TheAnalysers

Asha Joshi, Dibyojyoti Sanyal, Thomas Glaser

Abstract

JavaScript is widely used for writing client-side web applications and is getting increasingly popular for writing mobile applications. JavaScript is over fifteen years old; nevertheless, the language is still misunderstood by what is perhaps the majority of developers and designers using the language. One of the most powerful, yet misunderstood, aspects of JavaScript are functions. The ability of defining a function inside another function and passing function as an argument of another function provides more flexibility to the programmer to build higher order function which intern supports higher abstraction. But misuse of this prower can lead to bad coding standard. While terribly vital to JavaScript, their misuse can introduce inefficiency and hinder an application's performance. The Googles V8 JavaScript engine nested version (functions inside other functions) is 42% slower than the equivalent version without this nesting. Thus we present results of a dynamic analysis aiming to detect these functions which can be hoisted and defined outside the inner function it is declared. This helps reducing unnecessary stack space. Unlike C, C++, and Java, there are not that many tools available for analysis and testing of JavaScript applications. In this paper, we present a dynamic analysis for identifying these nested function (hoistable functions) using simple yet powerful framework, called Jalangi[1]. Other than reporting the hoistable function out analysis also results the total number functions that were executed, not executed, not hoistable and outer functions. Further the analysis is evaluated by running on well known libraries of node.js namely underscore[2], lodash[3] and commander[4]. We believe that all no false positives were reported; hence the analysis is quite efficient and effective.

Keywords

JavaScript; Dynamic Analysis; Jalangi2; hoistable function; nested function;

1. Introduction

Today's JavaScript engines are light-years ahead of the engines of ten years ago, but they do not optimize everything. What they don't optimize is left to developers. This is pushing JavaScript developers to ensure their code is efficient and has a good performance. The problem with nested functions is one characteristic of JavaScript that hinders performance: the nested function is repeatedly created due to repeated calls to the outer function. The misuse of functions can introduce inefficiency and hinder an application's performance. The Googles V8 JavaScript engine nested version (functions inside other functions) was reported 42

The ability to define a nested function addresses new programming construct which provides more flexibility and power to programmer. but if misused the advantage it provides can turn to a disadvantage. nested function helps programmer to use inline function call and to define singleton function which has their own advantages. It also provides ability to create higher order functions which can be passed as arguments and returned as value. This new programming construct is useful when we need more abstraction and reuse in lower level. Using function expressions to define a function inside another function gives the ability to define a function dynamically and determine which function to use programmatically. The ability to access a variable defined in the same scope an inner function is defined promotes the programming construct known as clause. Clauses also have an advantage in web based programming. But it depends on the situation which programmer has to decide whether to use nested functions. It is a performance overkill both in terms of performance and memory consumption if the nested function are defined in a loop or defined inside a function which is called several times.

Further paper is organized as follows. Section 2 discusses various scenarios of nested functions which can be hoisted including a hoistable function identified in commander library. Section 3 describes the implementation details. Further in Section 4, the evaluation technique for analyzing the three libraries and analysis results are presented. Section 5 pinpoints limitations and future work. Section 6 concludes the paper.

2. Scenarios

Before creating a functionality or logic it is important to understand all possible requirements and scenarios. Thus the first approach was to identify all possible scenarios where nested function can exist. As a result we identified 10 different scenarios ranging from normal blocks, Nested function declarations, more than one levels of nesting and different scopes. In Listing 1, function `add5(x)` is hoistable as it does not use any values from the parent function add. The function `addB()` is not hoistable as it uses variable b of parent function `add()`. However, since it is dynamic analysis and ideally `addB()` function will never be executed so, it will also not be analyzed. So, for this code snippet our analysis will result function `addB()` cannot decide if can be hoisted and `add5()` can be hoisted.

Some special cases where also identified where function is defined using an expression. In Listing 1, the function at line `1` is actually an anonymous function (a function without a name). In Listing 1, variable `b` is passed to function at line `1` and is again initialized with new values in inner function `add5()`. Function `add5()` is reported as can be hoisted. In npm library commander one hoistable function was identified. The Listing 2 shows this function. This was an anonymous function declared inside another anonymous parent function. Here the inner function is not dependent on the parameters of parent function and hence, can be hoisted.

[1] https://github.com/Samsung/jalangi2

[2] https://www.npmjs.com/package/underscore

[3] https://www.npmjs.com/package/lodash

[4] https://www.npmjs.com/package/commander

Listing 1. Example Function with nested functions

```
var add = function(a, b) {
    function addB(x) {
        return x + b;
    }
    function add5(x) {
       var b = 5;
        return x + 5;
    }
   if (b === 5) {
        return add5(a);
    } else {
        return addB(a);
    }
};

add(3,5);
```

Listing 2. Hoistable function found in commander

```
Command.prototype.action = function(fn) {
  var self = this;
  var listener = function(args, unknown) {
    args = args || [];
    unknown = unknown || [];
    ...
    ...
    fn.apply(self, args);
  };
```

3. Implementation

3.1 Jalangi2

The implementation of the analysis for detecting hoistable functions in JavaScript programs is based on Jalangi2. Jalangi is a proven dynamic analysis framework for JavaScript. Jalangi generates an instrumented intermediate code from JavaScript which it used for its analyses. It assigns a unique id `sid` to each JavaScript loaded in runtime. Each `sid` is mapped to an object called `iids`. `iids` is an array which represents each instruction in the code using a `iid` which stands for instruction id. The `iid` is a unique value which is assigned to each callback function inserted by jalangi. `iids` also maps each `iid` to an array containing `[beginLineNumber, beginColumnNumber, endLineNumber, endColumnNumber]`. The mapping from `iids` to arrays is only available if the code is instrumented with the `--inlineIID` option.

Jalangi has several types of callback functions as mentioned above. Each instruction in a program can be mapped to a callback. While executing the target program using jalangi the analysis executes the callbacks for each of the instructions. To test a particular behavior of the program the callbacks can be used in the analysis and can be implemented as needed. Jalangi analyses can be executed online using browser or using command line. Our analysis runs in a command line where the written analysis program and the program on which tha analysis is to be performed has to be specified with few other optional parameters to the Jalangi executor program.

Now we will discuss in brief the callbacks we used in our analysis. Each function with other parameters provides a -lstinline—iid—parameter which can be used to uniquely distinguish the instruction for which the callback is called. Our analysis logic uses the below callbacks. `functionEnter`, `functionExit`, `declare`, `read`, `write` and `endExecution`.

functionEnter(iid, f, dis, args) Function enter call back is called when a function body is about to start at the time of unction execution. parameter `f` provides the function object whose body is going to be executed. The `name` attribute of the function object can be used to to retrieve the function name. Using the `iid` parameter the line number of the source code where the function is declared can be obtained.

functionExit(iid, returnVal, wrappedExceptionVal) Similar to `functionEnter` `functionExit` is called when the execution of a function body completes.

declare(iid, name, val, isArgument, argumentIndex, isCatchParam) This callback is called for every local variable declared in the scope, for every formal parameter, for every function defined using a function statement, for arguments variables, and for the formal parameter passed in a catch statement. `name` provides the name of the variable declared and `val` provides the initial value of the variable at declaration time. `isArgument` parameter is `true` if the variable is a arguments or a formal parameter. `isCatchParam` parameter is `true` if the variable is a parameter of a catch statement.

endExecution() This callback is called when an execution terminates in node.js.

read(iid, name, val, isGlobal, isScriptLocal) Read call back is called after a variable is read. `name` parameter provides the name of the variable being read. `val` parameter provides the value of the variable. `isGlobal` is `true` if the variable is not declared using var keyword in the current script. `isScriptLocal` parameter becomes `true` if the variable is declared in the global scope using var.

write(iid, name, val, lhs, isGlobal, isScriptLocal) Write callback is called before a variable is written. the meaning of the parameters are same as read callback.

3.2 Implementation logic

During the execution the analysis reads through the instrumented object program and notes down the functions which are declared using function statement. The declare call back is used for this purpose. It is checked if the function is uniquely declared or not. The uniqueness of the function is checked with its name and its parent name. if the function is not declared before with same name and parent it will be noted and kept in a array. Write call back is used instead for the function which are declared and stored in a variable in `var v = function foo(){}` style. Similar checks are done as read callback for checking uniqueness of the function declared. The uniqueness checking is required because if outer function is executed several times the inner function declared inside should be checked once and should have an entry once in the array.

The function name with the line number of the function can be stored. Function line number is preferred because often we encounter anonymous functions. An anonymous function always called inline. Therefore there will be no conflict which inner function inside a outer function is executed. An unique id is maintained for each unique function declaration encountered before its is sored in the function array. The array stores all the details regarding each function.

When a variable declaration is encountered it is kept in a map which contains the variable and the function name under which the variable is declared. Here we skip the variables which are used to declare a function and the variables which are catch parameters. An unique id for each function is also maintained in the map. Now

when a function is called and it is executed the `functionEnter` callback is fired. This callback is used to maintain a stack. The stack holds the execution tree of functions. If a parent function is executed it is pushed in the stack. When the inner function of the parent function is executed the inner function is pushed on top of the stack. Using the stack it can be inferred which function is the parent function of which inner function. The top most function in the stack is always the current function in execution. As at a time only one function is in execution there is no chance of confusion if an outer function has multiple inner functions. Or if the name of the inner function is matched with an inner function in another outer or parent function. Even if the there are two parent functions with same name and where their inner function names are also same this will not raise any conflict, because the parent functions will be in different scope (as we know two function can not have same name in the same scope). The `functionExit` callback is used to pop the function from the stack and the all the function and variable pairs from the map when executing of a function ends. This approach ensures that even if there are several inner functions declared and executed under a outer function the top most function name in the stack is always the inner function of the second function from top in the stack. This technique also helps to clean the stack and map and gets rid of unwanted entries and keep them precise. When a function is in execution and a variable is read or written a check is performed to verify whether the variable is a variable which was declared in the parent function or any of the ancestor functions. This can be caught using read callback. If a variable is read in the outer most function in the program there will be only one entry in the stack and we can refer them as a global variable for which this check will be skipped. We also skip checking for the variables which are declared locally in the same function. If there are two variables in parent and inner function with same name the local variable will be in effect. This scenario can be avoided using this check. If a variable is not declared locally in any function the map containing function name and variable name is searched. if there is any entry found in this map with any function name which not the current function in execution that shows the function is either the immediate parent or any ancestor of the function in execution. There is no chance of referring an outdated function for which the execution is finished because as soon as execution of any function ends all the entries for the function is removed from the map. If such a variable is found the function name in top of the stack is picked which is the inner function in execution. And it is searched in the array of functions. And the function is annotated with the variable name and the parent function name under which this function is declared to identify that this function can not be hoisted. The second top most function in stack is always the direct parent of the inner function which is detected as non hoistable. If the variable was not declared in the direct parent but in any of its ancestors this also can be detected. In this case the second top function in the stack and the function name corresponding to the variable in map will not be same. If no such variable is found for a function when the function execution is ended `functionExit` callback is used to specify that the function is hoistable.

A nested hoistability check is done to make sure if any inner function is not hoistable making the outer function containing it also non hoistable, provided the outer function is declared inside respective ancestor. Respective ancestor is the function whose variable has been used in the inner function. After execution in `endExecution` callback nested hoistability is checked and `funcHoistableMap` is used to write a report in text file in easily understandable format.

If no parent information is found in the function array for any function that means the function was outer most function for which hoistability can not be applied. If a function is flagged as non hoistable and there is a parent function name associated with it, it has been written in the report as the function is non hoistable for the variable which is declared in its direct parent. If a function is flagged as non hoistable and there is no parent function name associated with it but it has an ancestor and a variable that means the function is non hoistable for the variable which is declared in its ancestor function. This is specified in the report. A function can be flagged as non hoistable and there is no parent function name or ancestor name associated with it and also there is no variable associated with it for which it can be told as hoistable. This scenario represents all functions which can not be hoisted because of the nested hoistability property. If any of the inner function of a function is non hoistable then the outer function itself becomes non hoistable when the inner function uses any variable which are not declared in the outer function but in the parent or ancestor of the outer function. Finally because in dynamic execution it may happen that a function is declared but not executed hoistability can not be decided for that function. And those are also written in the report.

3.3 Implementation details

The Stack The stack is called `funcNameStack` which stores all the function names when any function declaration is encountered.

The Map The map is called `funcVarMap` array which contains `{function id, function name, variable}` pairs, to store variables declared in a function.

The Array `funcHoistbleMap` array is the main array to store all information about all the functions, we store the below information for each function.

- A unique id for every function
- function name
- hoistable - flag which denotes if the function is hoistable
  - false - if cannot be hoisted
  - true - if function can be hoisted
  - undefined - denotes dynamic analysis was not able to identify hoistablility (path did not execute)
- parent function under which this function is declared(NoParent - for outermost functions)
- parent variable (if any)due to which function is not hoistable.
- respAnscestor - function who's declared variable is being used (might not be immediate parent)

Figure 1 shows the values collected during the execution of the first example.

4. Evaluation

We tested our analysis with node version v0.10.37 on an ubuntu 14.04 LTS operating system. The analysis did run on an intel core i7 4 x 1.9GHz with 4GB RAM and took under a minute to compute on every test.

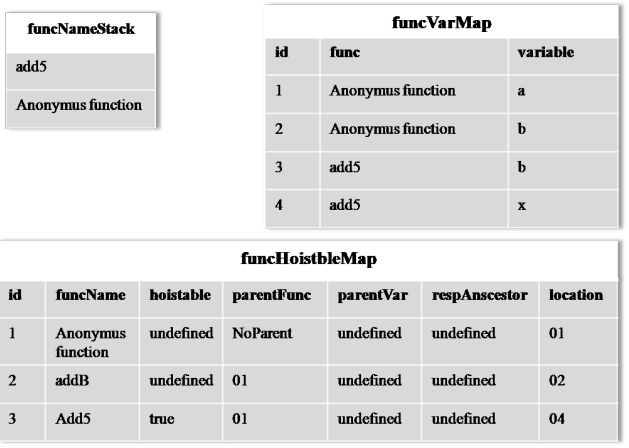
We tested our analysis on ten own created scenarios we wanted to address. All of them behaved like expected and covered all different possibilities what could make a nested function hoistable or not. I addition we tested our analysis on three of the most dependet upon packages of npm[5], underscore, lodash and commander. We modified their respective test suites based on the QUnit[6] testing

[5] https://www.npmjs.com/browse/depended

[6] https://qunitjs.com/

package	total # functions	# not executed	# not hoistable	# outer most	# hoistable
underscore	24	23	0	0	1
commander	14	1	0	12	1
lodash	813	638	19	0	156

Table 1. Evaluation on the three libraries.



funcNameStack

add5
Anonymus function

funcVarMap

id	func	variable
1	Anonymus function	a
2	Anonymus function	b
3	add5	b
4	add5	x

funcHoistbleMap

id	funcName	hoistable	parentFunc	parentVar	respAnscestor	location
1	Anonymus function	undefined	NoParent	undefined	undefined	01
2	addB	undefined	01	undefined	undefined	02
3	Add5	true	01	undefined	undefined	04

Figure 1. Collected values during execution of first example.

framework to execute the jalangi instrumented version of the libraries. With this we were able to analyze the libraries without taking the execution of the testing suites itself into account. Finally we integrated all of the commands to run our analysis into one script that analyses our own tests and the tests of the libraries and merges the outputs into one big output file. For each function we encounter we decide, whether it is hoistable, not hoistable or not decidable. If the nested function is hoistable, we provide the line number in the original code to make it easier to find the function. If a nested function is not hoistable we give the reason why. If the function is only declared but not executed we can not decide due to our dynamic approach.

For all three libraries we were able to identify hoistable and not hoistable functions. Table 1 shows our results. In underscore we found out of the 24 functions declared 1 hoistable function, for the other functions we were not able to decide whether they are hoistable or not. In commander 12 of the 14 functions were the outer most functions, one was not executed and the last one is the hoistable function used as an example. Lodash hat a total number of 813 functions, for which are 156 hoistable. 19 are not hoistable due to used variables and the other 638 the analysis could not decide whether tey are hoistable.

The high number of not decidable respectively not executed are functions, that have no source code in the actual library, like native functions, and therefore can not be analyzed during execution, so they get considered ”not executed”.

Another point to discuss are functions that are nested in one big directly executed anonymous outer function. They are technically nested functions, but the outer function is only executed once and is determined to be an outer function. You could consider them not as nested functions, but our approach does not take this into account.

5. Limitations and future work

A variable declared anywhere in the program without using var is a global variable. This can not be identified in Jalangi2. Ideally this should be caught under read callback and skipped using `isGlobal` or `isScriptLocal` parameter, but jalangi fails to identify the variable as global variable.

The current analysis checks a function for hoistability based on the usage of parent function parameters. However, the variable declared inside the parent function are in case if used in child function then child function will still be reported as hoistable. In Listing 2, the function at line 3 is reported as hoistable with current analysis implementation even though is it using the parent variable `self` declared at line 2.

6. Conclusion

We have developed a dynamic analysis which was tested on three node.js libraries. The hoistable functions detected in all cases where checked manually in the source code and no false positives were found. Our analysis provides additional information on the functions which were not executed, cannot be hoisted and which are outer most functions.