Hoisting Nested Functions

Anam Dodhy
Informatik, TU Darmstadt
anam.dodhy@stud.tu-darmstadt.de

Chaitra Hegde
Informatik, TU Darmstadt
chaitra.hegde@stud.tu-darmstadt.de

Abstract

With the widespread use of Javascript in web development, performance is a factor of concern which may affect the speed of the browser. One such performance limiting factor is nested functions in JavaScript. In this project, we have developed an analysis using Jalangi2 [Ref:: Jalangi] framework and NodeJS to detect the hoistability of the nested functions in Javascript. This approach is evaluated on the Javascript Library *Lodash*, to check the improvement in the performance of Javascript in Google V8 engines after hoisting nested functions.

1. Introduction

JavaScript is used for both client and server side scripting. Among the many features it provides, one of the features that we are interested in is hoisting of function declarations. *Hoisting* is moving the variables or function declarations outside of the scope they are currently residing in. JavaScript also allows nesting one function inside the other. This is because nesting reduces the global namespace pollution[Ref:: 1]. Also, if a function has to access the local variable of another function then it has to be nested under the function whose variable it wants to access. This helps in maintaining the privacy of data inside the function.

The nested (inner) function is created only when the parent (outer) function is invoked. So if the outer function is called in a loop then the nested function gets created on every single call no matter if it is used for the computation or not. This would reduce the performance of the Javascript engine. So if the nested functions are not dependent on any variables or parameters of the parent function, it is better to hoist them to improve the performance.

In this project we are using Jalangi2, a dynamic analysis framework for JavaScript to detect if a function is nested and if it can be hoisted or not. Jalangi2 instruments a given piece of Javascript code and provides many APIs or callbacks to check what values are exchanged and how the control flow of the JavaScript file is in the interpreter. We are using functionEnter(), functionExit(), read(), write() and declare() to first identify the nested functions and then later determine whether any of those nested functions are hoistable or not.

2. Hoisting Rules for Nested Functions

To hoist a function declaration to another scope, we need to follow these two rules:

Rule 1: The nested (inside) function must not depend on any of the local variables of the outer (containing) function/s. These variables include both local variables defined in outer function and parameters of the outer function.

It makes sense to not move a function declaration, which is using the parent's local variable/s, to the outer scope of the parent, when the nested function's sole purpose is to maintain the data privacy of the parent function. For example, in Listing 1 we cannot hoist "functionChild3()" because it uses the parameter "a" of its parent "functionParent(a)". Also, we cannot hoist "functionChild4()" because it uses the local variable "b" of its parent function which is again "functionParent(a)".

```
1  function functionParent(a) {
2    var b = 1;
3
4    function functionChild3() {
5      var z = a + "Child3";
6    }
7    function functionChild4() {
8      b = b + "Child4";
9    }
10   function functionChild2() {
11     var x = "Child2";
12
13     function functionChild3(){
14         var y = 8;
15         return
16     }
17     return functionChild3();
18   }
19  functionChild4()
20  functionChild3()
21  return functionChild2();
22 }
23 functionParent("Hello");
```

Listing 1. Nested Function Declaration Example

Rule 2: If Rule 1 is satisfied, then the nested function definition can be hoisted. But, then there should be no other function definition with the same name as the nested function at the scope where the hoistable function has to be moved.

In Listing 1, nested function "functionChild3()" inside the scope of "functionChild2()" satisfies Rule 1 as it is neither dependent on the local variables nor the parameters of its parent function "functionChild2()". But we still cannot hoist the nested functionChild3() outside of its immediate parent's scope because there is already a function with the same definition at that level. Hence, Rule 2 would be violated if functionChild3() is hoisted under "functionParent(a)". But we can still hoist this function one step above the "functionParent(a)" which would be at the root level since at this level we do not have any function definition conflicts. Therefore, we can deduce that functionChild3() under functionChild2() can

```
Node{
  name,
  parent,
  funcBody,
  children[],
  variables[],
  isHoistableWithParent,
  nonHoistableParents[],
  iid
}
```

Listing 2. TreeNode Structure

be hoisted just not under functionParent(). Result of our analysis for this example can be seen in Figure 5.

3. Algorithm of the Analysis

As mentioned in the Section 1 we are using Jalangi2, a javascript dynamic analysis framework for our project. This framework instruments any given piece of Javascript code by attaching or hooking up its callbacks to it. In this project, we are using following Jalangi2 callbacks: [Ref:: Samsung Jalangi2]

- *functionEnter():* Triggered before starting the execution of a function
- *functionExit():* Triggered when the execution of a function is completed
- *read():* Triggered after reading a variable
- *write():* Triggered before writing a variable
- *declare():* Triggered when the scope of a local variable starts
- *literal():* Triggered upon creation of a literal which can be an array, boolean, number etc.
- *instrumentCodePre():* Triggered before a given function is instrumented or before an eval() gets string as an argument.

FunctionEnter() and FunctionExit() are the two basic callbacks which define the entry and exit point of our analysis. Moreover, in order to keep track of the nested function hierarchy, we are maintaining a simple Tree in which each "node" represents all the information required for a single function to determine its hoistability.

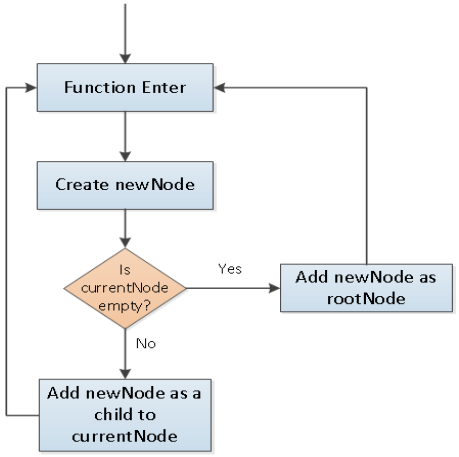

Figure 1. Approach Entry Point

Listing 2 shows the data structure of each node where each "node" represents a function and in it a "*name*" stores the actual name of the function. "*parent*" contains the parent node of each node, and "*children*" points to all the children nodes of a given node. "*funcBody*" contains the actual function body of the node which is later used in the analysis to determine recursive function calls (Section 4.2). "*variables*" is an array of objects where each object contains the variable name, its type i.e. declare, read or write and a flag to determine if it was an argument or not. "*isHoistableWithParent*" is a boolean flag which is set to true if Rule - 1 of hoistability for a given node is satisfied. "iid" contains the line number of the function in the source code. Lastly, "*nonHoistableParents*" is an array of all the parent nodes under which the given node cannot be hoisted due to function definition conflicts (Rule-2).

Figure 1, is a flowchart diagram of the entry point of our analysis. So when a function is called, the FunctionEnter() callback is invoked in which we create a new node for the function being called. After creating the node, the analysis checks whether the "currentNode", which is the node representing any function called previously, is empty or not. If it is empty then it is assumed that this is the very first function call of the code base and hence, the newNode is to be added as the "rootNode" of the analysis tree. On the other hand, if the currentNode is not empty, then the analysis would rightly detect a function nesting and would add the newNode as a child node to the currentNode. In both cases, the currentNode is updated at the end to newNode.

Next read(), write() and declare() callbacks are invoked after the functionEnter() callback finishes. These callbacks are used to check for all the parameters and variables declared or used by the function (currentNode) and then add them to the array of "variables" shown in Listing 2. Lastly, "instrumentCodePre()" is used to know if an eval() is direct or indirect and "literal()"is used to determine the name of the function in case of of an indirect eval().

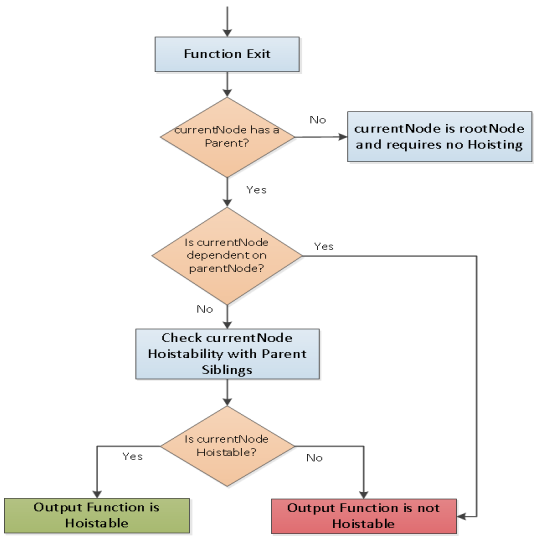

Figure 2. Approach Exit Point

This is how the whole tree structure is built for all the functions by the analysis. Now, when a function represented by "currentNode" in Figure 2 exits then the functionExit() callback is invoked. In which case the analysis first checks Rule-1 (Section 2) of hoistability for the function (currentNode). This means that the analysis checks whether currentNode is using any of the variables or parameters defined or used by its parentNode. If yes, then the flag "isHoistableWithParent" is set to *false* and at the end, this function would be marked as "Not hoistable".

```
functionEnter(functionObject){

  TreeNode newNode =
  new TreeNode(functionObject,currentNode)
  if(currentNode==null){
    tree.push(newNode)
  }
  else{
    currentNode.addChild(newNode)
  }
   currentNode = newNode
}

functionExit(){

  checkHoistabilityWithParent(currentNode)
  if (currentNode!=null &&
  currentNode.parent!=null){
  currentNode=currentNode.parent
  }
  else if(currentNode.parent==null){
     checkHoistabilityWithParentSiblings(
    currentNode)
  }
}
```

Listing 3. Pseudocode of the Analysis

On the other hand, if the function (currentNode) is not dependent on its parent function (parentNode) then this flag "isHoistableWithParent" is set to *true*. Now comes the second rule of hoistability (Section 2) which is to make sure that the scope in which the function (currentNode) is to be hoisted does not already have a function with the same definition. To verify this rule for a given function referred as currentNode in Listing 3 and Figure 2, this analysis recursively traverses through the whole tree hierarchy above the currentNode and adds all the parents which violate this rule to the list of "nonHoistableParents" of the currentNode. Thus, in the end, the analysis would output exactly under which parent the currentNode would face a definition conflict problem.

4. Completeness of the Approach

The above mentioned approach works fine for the function declarations. In the below subsections, we are checking the coverage of our approach in different use/corner cases:

4.1 Function Expressions

When a function definition is assigned to a variable, it is called a *function expression*. The function assigned to the variable can have a name or it can be "anonymous", i.e. without a function name.

For a given code such as in Listing 4 with function expressions, we first check if the nested function expression is following hoistability Rule-1. If it is, then it is marked as hoistable and can be assigned to a new variable in the outer scope. For example, in Listing 4, where functionTwo() and functionThree() are nested inside functionOne() and functionTwo() respectively. Our approach will first apply Rule-1 on all the functions, functionTwo() is not hoistable because it uses the local variable "param" of its parent functionOne(). FunctionThree() is marked as hoistable by our analysis because it is not using any local variables of functionTwo() and then our analysis will check hoistability Rule-2 for this function and identify any function definition or variable name conflicts with the parent function.

```
var one = function functionOne(param) {
  var b=1;
  var two= function functionTwo(){
     var v = param +1;
     var three = function functionThree(){
            var c = c + 1;
            return c;
         }
         return three()
      }
  return two();
}
one(2);
```

Listing 4. Function Expression

In case of anonymous functions, which are basically function expression without any function names, our approach stores the function name as "anonymous" while "creating a new node" shown in Figure 1. For the rest of the steps, anonymous functions are treated as normal function expressions by our analysis. Figure 3 shows the our analysis output for Listing 4.

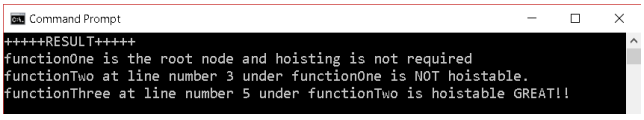

Figure 3. Analysis Output for Listing 4

4.2 Recursive Functions

While adding a child to a node, we compare the name and function body of the parent function with the child function. If they match and the names are not "anonymous", then we mark the function as *recursive* and we do not add it as a child function to our Tree structure again. In this way, the recursive function is only added to our Tree structure once.

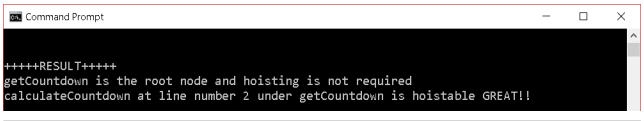

Figure 4. Analysis Output for Listing 5

Consider Listing 5, this piece of code just counts down from the given input. Here if we print the output, it should give us the value 5,4,3,2,1. Function calculateCountdown() is nested inside getCountdown() function and it is not using any local variables of the parent function. So our approach will only add calculateCountDown() as a child node to the getCountDown() function once, and then later upon functionExit() will check both hoistability Rule-1 and Rule-2 and correctly mark it as hoistable. Figure 4 shows our analysis's out for Listing 5.

4.3 Direct and Indirect eval()

A javascript program can have a function defined inside eval(). Our approach takes care of this case as well. *eval()* takes the scope where it is invoked. If the function is invoked inside an eval(), then it takes the scope where it was invoked. Our approach applies Rule-1 and Rule-2 to check the hoistability of a function inside an eval().

In Listing 6, directEval() is in the scope of funcEval(). So it uses the variables "x" and "y" of funcEval() and gives the output 6.

```
function getCountdown(countdownValue){
 function calculateCountdown(value) {
   if(value>0) {
     return calculateCountdown(value-1);
   }
   else{
     return value;
   }
 }
    calculateCountdown(countdownValue);
}
getCountdown(5);
```

Listing 5. Recursive Functions

```
x = 3;
y = 5;
function funcEval() {
 var x = 2;
 var y = 4;
 eval('function directEval() { return x +
    y; }');
 return directEval();
}
funcEval();
```

Listing 6. Direct eval()

directEval() cannot be hoisted according to the hoistability Rule-1 and our analysis shows the same output as well.

Now when an eval() is assigned to a variable, it is called an *indirect eval*. The scope of an indirect eval() always takes the global scope no matter where is it invoked. So a function invoked using an indirect eval() is always hoistable even if it is invoked inside another function's scope.

In Listing 7, "geval" is the reference for eval and it is invoking a function at line 7. Though indirectEval() at line 8 appears to be nested inside funcEval(), but it will not use any variables of this scope i.e. "x" or "y" at line 5 and 6 respectively. Instead it will use the global variables "x" and "y" at line 1 and line 2 for calculating the return value and will return 8. Our analysis handles this case by first detecting the presence of an indirect eval during the instrumentCodePre() and literal() jalangi2 callbacks. We store the function names in an array. In Listing 7, we store inDirectEval(). While creating a child node, we do not include inDirectEval() to the current node's children. But we display it in the result that since inDirectEval() is indirect eval, it does not require hoisting.

In cases where the functions defined inside direct or indirect eval() are nested, our approach successfully handles the hoistability detection by treating those functions as normal functions declarations or expressions only with different scopes.

5. Evaluation and Results

In order to evaluate the accuracy our analysis, it was first executed on smaller but complicated Javascript programs like the ones shown in Listings 7, 6, 4, 5. The analysis was able to successfully detect all hoistable functions by checking both Rule-1 and Rule-2 on such examples.

For example, Figure 5 is the screen shot of the output our analysis gave when we executed it on the code given in Listing 1.

```
x = 3;
y = 5;
var geval = eval;
function funcEval() {
  var x = 2;
  var y = 4;
  geval('function indirectEval() { return
    x + y; }');
  return indirectEval();
}
funcEval();
```

Listing 7. InDirect eval()

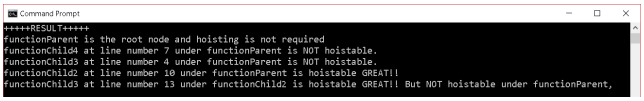

Figure 5. Analysis Output for Listing 1

Next phase of the evaluation was focused on detecting any difference in execution time of a code base with hoisted functions and without any hoisting. To do this a famous Javascript library "Lodash" was selected and following steps were performed:

1. First, the provided test suit of the Lodash library was executed on the original version of the repository multiple times and an average execution time was noted down.
2. Then the analysis was executed on the same repository which marked all the hoistable and non-hoistable functions.
3. Then as per the analysis results, manual hoisting of the repository was performed.
4. In the end, the Lodash test suit was again executed multiple times but this time on the updated hoisted version of the repository and an average execution time of the test suite was noted down.

Following are the results which were obtained:

Table 1.

Library	Avg. Execution Time	
	Original Repository	Hoisted Repository
Lodash	9767ms	9704ms

So it can be seen from the results in Table 1, that after only hoisting 7 functions an improvement of around 63ms was observed in the execution time.

6. Limitations and Future Work

Our approach has few limitations which can be used for future work. Firstly, while checking recursive functions, we just compare the names and function body of the parent function with the child function. But in practice, Javascript allows for nested functions to have the same same name and function body as its parent function, in such a case our approach would detect that as a recursive function as well which would be incorrect.

Secondly, if there is a nested function which is not dependent on any of its parent variables or parameters and has no function definition conflicts with its parent's siblings but calls a function defined in the same scope as itself. For such a program our approach just checks the two rules of hoistability and would mark it as hoistable

even though when hoisted, this function would throw an exception as it was calling a function inside it which is no longer accessible in its new hoisted scope. This is something that can be improved in future.

Moreover, our analysis tells exactly under which parent a function can't be hoisted but it does not tell under which parent hoisting the function would be most optimal. Such kind of information can be helpful in the case where there are multiple options of hoistability for a particular function and hoisting the function at one place might give better performance than the other. Plus, in case of indirect eval, our anaylsis is unable to detect hoistability of any nested functions defined inside of the indirect eval itself.

7. Conclusion

Nested functions have to be avoided when it does not serve the purpose of abstraction and encapsulation of data. Also, nesting functions reduces the performance. So it is better to identify such functions and hoist them. Our approach uses two basic rules of hoistability to identify if a function can be hoisted or not and except for some limitations described in Section 6 we are successful in identifying functions which can be hoisted. Also, evaluation of approach on javascript libraries like Lodash showed positive results by showing improvements in execution time.

References

. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 488âĂŞ498. ACM, 2013.