Hoisting Nested Functions

Feras Tanan

Listing 1. Nested Function Definition

```
1 function outerFun(x,y,z){
2         function innerFunc(a){
3                 return a*a;
4         }
5         return x+y+z+innerFunc(2);
6 }
7 outerFun(1,2,3);
```

Listing 2. Hoisted Version of Listing1

```
1 function innerFunc(a){
2         return a*a;
3 }
4 function outerFun(x,y,z){
5         return x+y+z+innerFunc(2);
6 }
7 outerFun(1,2,3);
```


Abstract

In this report, we produce a dynamic analysis approach which extracts all function definitions that can be hoisted using dynamic analysis framework Jalangi framework. This approach was evaluated on the following JS Libraries: Q_1, Underscore and Lodash. The accuracy of this approach was 100%, 50%, and 100% respectively.

Keywords: Hoisting Functions - Nested Functions- Dynamic Analysis.

1. Motivation and Introduction

JavaScript allows the developer to define functions inside other functions. This has a significant performance drawback. Every time the containing function is called, the inner function is created. If we have a JavaScript program which has two functions defined in a nested way. This JavaScript program has slower execution time than it's hoisted version; another version of the same program with the two functions defined at the same level [not nested]. Listing 1 shows a JavaScript program that defines two nested functions and Listing 2 shows the hoisted version of the same JavaScript Program.

The goal of this project is to create a dynamic analysis tool that is able to analyze a JavaScript program in order to find a list of all functions that can be hoisted. In other words, the dynamic analysis approach must find all function definitions that:

- are defined inside other function definitions .
- can be defined outside the surrounding function definition.

The rest of this report discusses the approach we followed to implement the dynamic analysis from two points of views: data structure and algorithm. After that we provide an evaluation of our work.

2. Approach

2.1 Hoisting Rules:

There are two basic Rules which tell when a function definition can be hoisted. These rules are:

- **Rule 1:** The inner function must not depend on the local variables of the containing function (parameters or defined vars).

  Listing 3. Rule 1 elaboration

  ```
  1 function outerFun(x,y,z){
  2         function innerFunc(){
  3                 return x*x;
  4         }
  5         return x+y+z+innerFunc();
  6 }
  7 outerFun(1,2,3);
  ```

  Listing 4. Rule 1 elaboration

  ```
  1 function outerFun(x,y,z){
  2         r= y-z;
  3         function innerFunc(v){
  4                 return v*r;
  5         }
  6         return x+y+r+innerFunc(2);
  7 }
  8 outerFun(1,2,3);
  ```

  In Listing 3 we can not hoist *innerFuncc* because it uses the parameter x of *outerFunc*. In Listing 4 we can not also hoist *innerFuncc* because it uses the local variable r of *outerFunc*

- **Rule 2:** when a function definition can be hoisted, we should make sure that there is no other function definition at the same level of the containing function that has the same name. Listing 5 elaborates this rule.

  The *innerFuncc* in line 3 of listing 5 satisfy rule 1. It does not depend on local variables of the containing function but we Can not hoist it because there is another function definition (*innerFuncc* at line 9 at the same level of the containing function *outerFunc*.

2.2 Used Callbacks

We analyzed the program dynamically using Jalangi dynamic analysis framework. Jalangi provides a set of callbacks that make dif-

Listing 5. Rule 2 elaboration

```
function outerFun(x,y,z){
        r= y-z;
        function innerFunc(v){
                return v;
        }
        return x+y+r+innerFunc(2);
}

function innerFunc(b){
                return b*2;
}
innerFuncc(4);
outerFun(1,2,3);
```

ferent components of the analyzed program accessible. For Example, function definition component, defining a variable, writing a variable, and so on. Having an access to a function definition component provide the ability to get a list of the local variables of the this function, the name of this function and so on. here are a list of the callback that we used to build our approach's structure:

- *read:* for getting the name of the read variable.
- *write:* for getting the name of the defined/written variable.
- *functionEnter:* this callback means that JavaScript engine is about to execute a function body, it gets the function object whose body is about to execute.
- *functionExit:* This callback is called when the execution of a function body completes.

  Both *functionEnter* and *functionExit* are useful for scoping. In other words, we used these two callbacks for knowing which function definitions are inside which functions definitions.

2.3 Data Structure

we developed a data structure that make us able to navigate the scanned program in order to find a list of functions that can be hoisted. Figure 1 shows the built data structure corresponding for listing 6.

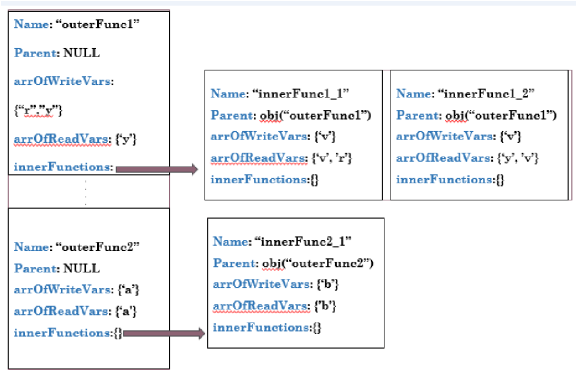

Figure 1. the built data structure of Listing 6

Interpretation of Data Structure: The data structure represented in figure 1 tells us that there is a definition for function called *outerFunc1* whose parent function is NULL and this function has both variables "r" and "y" as written/defined variables (we consider the parameter as written variable in order to make Rule 1 in section 2.1 applies also for inner functions definitions that depend

Listing 6. a JavaScript program to be analyzed

```
function outerFun1(y){
        r= y-2;
        function innerFunc1_1(v){
                return v*r;
        }
        function innerFunc1_2(v){
                return v*y;
        }
        return y+r+innerFunc1_1(2)+innerFunc1_2(3);
}
outerFun(3);

function outerFun2(a){

        function innerFunc2_1(b){
                return b*b
        }

        return innerFunc2_1(3)+a;
}
```

Listing 7. the algorithm which checks the data structure if it contains functions that can be hoisted

```
function find_hoisting_funs(function_list)
 foreach function f in function_list
   hoistable_funs.add(f)
   foreach variable var in read_vriables of f)
        if (var is defined in f)
        and
         (var is defined before reading its value)
                continue
        else
        if (var is passed to f as parameter)
                continue
        else
        if(var defined in the parent of f)
           hoistable_funs.remove(f)
           break
        if(f has inner functions)
           find_hoisting_funs(inner functions of f)
```

on the parameters of the containing function). *outerFunc1* has read one variable which is the parameter "y". It also has two inner function definitions $innerFunc1_1$ and $innerFunc1_2$. The data structure shows also which variables are written/read in these two inner functions and shows that the parent (containing function) of these two parameters are the object which handles *outerFunc1*. The same interpretation applies for *outerFunc2*.

2.4 Algorithm

We developed an algorithm which traverse the previous explained data structure in order to find a list of all the function definitions that can be hoisted. listing 7 shows a pseudo code of this algorithm.

2.5 Corner Cases

The following corner cases have been treated successfully in our approach

Listing 8. corner case 1 ”indirect eval”

```
1 var indirecteval = eval;
2 var a = 1;
3
4 function f(a) {
5  indirecteval( function g() {
6  return a + 1;
7  }   );
8  return g();
9  }
```

Listing 9. corner case 2

```
1
2 function outerFunc()
3 {
4   x=7;
5   function innerFunc()
6   {
7    x=4
8    return x*2
9   }
10  return x+innerFunc();
11 }
```

Listing 10. corner case 2

```
1
2 function outerFunc()
3 {
4   x=7;
5   function innerFunc()
6   {
7    v=x+2
8    x=5
9    return v
10  }
11  return x+innerFunc();
12 }
```

corner case 1: JavaScript provides ”eval” instruction which takes a string and executes it as JavaScript code in the current scope. If the value of eval is assigned to another variable, it instead becomes indirect eval, evaluated in the global program scope instead of the local o function scope. In Listing 8, the function g can be hoisted because it is in an indirect eval. JavaScript executes g in the global scope. g in Listing 8 access variable a defined in line 2.

corner case 2: if the inner function uses (reads) a variable defined in the current scope and in the parent scope then it can be hoisted. In Listing 9, innerFunc reads x which was defined in innerFunc and in outerFunc.

corner case 3:if the inner function uses a variable that is defined in both parent scope in current scope. But, in the current scope, it uses this variable before defining it then the the inner function can not be hoisted. Listing 10 shows that variable x is defined in both outerFunc and innerFunc. innerFunc uses variable x at line 7 before it defines it at line 8. This means that innerFunc depends on outerFunc and it can not be hoisted.

3. Evaluation

We evaluated this approach by trying the dynamic analysis which we developed on three JavaScript libraries. The result of executing the dynamic analysis on these three libraries are shown in table 1.

Interpretation of the results in table 1: for library Q1,out of 10 traversed function definitions, our tool was able to find 2 function definitions that can be hoisted. We examined Q1 source code manually and found that only the two function definitions which the tool found can be hoisted.

for Library Underscore, out of 12 traversed function definitions, our tool was able to find 4 function definitions that can be hoisted. We examined Underscore source code manually and found that only 2 of the four function definitions which the tool found can be hoisted. There are here two false positives.

for Library Lodash,out of 51 traversed function definitions, our tool was able to find 33 function definitions that can be hoisted. We examined Lodash source code manually and found that the same 33 function definitions which the tool found can be hoisted.

Library	Traversed Functions	Automatically	Manually
Q-1	10	2	2
Underscore	12	4	2
Lodash	51	33	33

Table 1. Evaluation Table

4. Conclusion

In this report we developed a dynamic analysis tool using Jalangi dynamic analysis framework. The tool takes as an input a JavaScript program to be analyzed and returns a list contains all the function definitions that can be hoisted. This approach depends basically on the idea that the inner function definition should not access local variables of the containing function definition in order to be hoisted.

We also presented an evaluation to this approach by testing it on the Libraries: Q-1, Underscore and Lodash.