Hoisting Nested Functions

Course Project proposed by Marina Billes
Program Testing and Analysis – Winter 2017/18

Introduction

JavaScript allows definition of functions inside other functions. These nested functions are created every time the outer function is called. Consider the following:

```
var x = 23;

function f(a) {
  function g(step) {
    return x + step;
  }
  g(a);
}
f(1);
```

Listing 1: Nested function definition

The function `g(step)` does not depend on the surrounding function `f(a)`. This nested version is 42% slower in Google's V8 JavaScript engine than the equivalent version with `g` defined outside of `f`, seen in Listing 2.

```
var x = 23;

function g(step) {
  return x + step;
}

function f(a) {
  g(a);
}
f(1);
```

Listing 2: Hoisted version

The function `g` cannot be hoisted if another function also named `g` already exists in the scope that `g` should be moved into.

Goal

The goal of this project is to develop a dynamic analysis to detect functions which can be hoisted. The tool should be based on the dynamic analysis framework Jalangi [2, 1]. For a function to be hoisted, it must not depend on any local variables of the containing function. Consider Listing 3.

```
function f(a) {
  function g() {
    return a + 1;
  }
  return g();
}
```

Listing 3: Nested function `g` depends on local variable `a` and cannot be hoisted

The approach should first find all nested function definitions. Then, it should identify nested functions which can be hoisted by checking whether it accesses local variables from its immediate outer scope or not.

Corner cases

In JavaScript, reflection mechanisms such as `eval` are commonplace. `eval` 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 function scope. In the following listing, function `g` is not nested in the scope of `f` and hence accesses the global variable `a` instead of `f`'s function parameter:

```
var indirecteval = eval;
var a = 1;

function f(a) {
  indirecteval(`function g() {
    return a + 1;
  }`);
  return g();
}
```

Listing 4: Usage of indirect `eval`

These corner cases need to be properly treated in order to provide correct hoistability information.

The dynamic analysis can only give an approximation: it needs to guarantee that for every execution it has observed, the function can be hoisted. Thus, functions have to be executed multiple times by appropriate test suites to check for their hoistability.

Tasks

More specifically, the project involves the following tasks:

- Get familiar with Jalangi and node.js[1], a tool for running JavaScript programs outside the browser.
- Create three simple node.js test programs which have nested functions that can be hoisted and nested functions that cannot be hoisted. Create a suite of tests to execute the functions several times.
- Design and implement an approach to detect function definitions which can be hoisted, using Jalangi. Test it with your test programs. In particular, check how complete your approach is. Does it work with function declarations as well as function expressions? Does it work with recursive functions and `eval`?
- Evaluate the approach by running it on node.js applications underscore[2], q[3] and lodash[4], which have test suites. After detecting hoistable functions, manually perform the hoisting and then compare the execution speed of the test suite with the hoisted version to the non-hoisted version.

Deliverables and Grading

The team must present the project between February 12 and 16, 2018, in a short talk, followed by a question and answer session. The exact time and date will be agreed on in due time. The team must submit the final project report and the project's implementation on February 25, 2018. The report and the implementation must be send to the project mentor via email (or, for large files, a file hosting service, e.g., Dropbox).

Grading will be based on the following criteria:

[1] `https://nodejs.org/`
[2] `https://github.com/jashkenas/underscore`
[3] `https://github.com/kriskowal/q`
[4] `https://github.com/lodash/lodash`

Criterion	Contribution
Report (structure, explanations, examples, writing)	20%
Approach (own ideas, independence, involvement, organization)	20%
Results (discussion and interpretation, soundness, reproducibility)	20%
Implementation (completeness, documentation, extensibility)	20%
Presentation (clarity, illustration, quality of answers)	20%

Reading material

[1] Samsung. Jalangi2: Dynamic analysis framework for JavaScript. URL `https://github.com/Samsung/jalangi2`.

[2] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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 488–498. ACM, 2013.