Documentation for *Project: One Analysis*

Collaborators: Judah Rowe, Maahee Abdus Sabur

Certification: Each person actively contributed to and fully understands this document.

# Introduction

The overall goal of our project is to develop a tool that statically detects dead code in programs. Our scope is a limited subset of JavaScript that includes variable definition/assignment, if statements, while loops, continue statements, break statements, try/catch blocks, and throw statements.

Our tool performs static analysis to detect two types of dead code: unreachable statements and dead assignments. For detecting dead assignments, we used the *Live Variable Analysis* described in *Principles of Program Analysis, Sec. 2.1.4* with slight modifications to fit the language we are analyzing.

# Analyses Implemented

## **Unreachable Statement Analysis**

The analysis problem for this analysis is: “Which statements must be unreachable on all possible control flow paths?”.

The property space is , where is the set of labels in the input program.

Notation: represents the program we are analyzing, represents the labels appearing in .

The mathematical formalization is:

The set includes all possible control flows in . If statement labeled is reached during ’s execution, one of two things is true: either is the initial label or there is a control flow that led to . If either of these things is true, will be excluded from . So, if a label is included in it is definitely unreachable.

This shows that the analysis is sound.

## **Live Variable Analysis (Nielson et al.)**

The analysis problem for this analysis is: “For each program point, which variables may be live at the exit from the point?”

The property space is , where is the set containing the names of variables in the input program.

Below is the mathematical formalization.

Notation: represents the program we are analyzing, is a set containing the arguments passed to the *console.log* function.

**Kill and gen functions**

**Data flow equations: LV=**

This is a may analysis. For it to be sound, if a variable is live at a particular point, the analysis must consider it to be live at that point. The dataflow equations ensure this by taking the union of live variables coming from all possible control flow paths.

# Architecture of Tool

Our tool is written in JavaScript. We read an input program, parse it to produce an AST, and analyze the AST to determine the flows of the input program. Here, we notify the user of any statements that are unreachable. We then compute gen and kill sets at each variable definition and reference. After that, we combine the gen/kill information with the flow graph to compute the live variables (LVentry and LVexit) at each point. With that, we notify the user of any dead assignments.

We use the *Acorn* library for parsing. Each node in the resulting AST has a “start” property which stores the location of the corresponding code in the input file. We use these “start” values as labels since they are unique for our purposes. We represent flows as an array of [a, b] pairs as in the textbook.

# References

Nielson, Flemming, et al. *Principles of Program Analysis*. Germany, Springer, 2010.

Shah, Aadit. “Comparing ECMA6 Sets for Equality.” *Stack Overflow*, 30 June 2015, stackoverflow.com/a/31129384.

“Acornjs/Acorn: A Small, Fast, JavaScript-based JavaScript Parser.” *GitHub*, github.com/acornjs. Accessed 14 Apr. 2024.