Static JavaScript Call Graph Construction

Jorryt-Jan Dijkstra
Software Engineering
University of Amsterdam
jorryt-jan.dijkstra@student.uva.nl

I. Introduction

Call graphs are indispensable graphs for programming languages.

II. Grammar

III. Call Graph Forms

IV. Data

V. Threats to validity

A. Javascript Grammar

Both JavaScript call graph algorithms require a parse tree as input. Code analysis at the CWI is often done with the Rascal meta-programming language, which provides a toolset that could help to ease the implementation of the algorithms. Rascal did not provide a JavaScript grammar to create a parse tree of a given JavaScript. Therefore creating a Rascal grammar for JavaScript interpretation was a prerequisite for the research project. The Rascal grammar seems to parse the given input scripts correctly and has been tested with unit tests with various snippets. Even though the grammar is thoroughly tested, it is possible that some deviations of the EcmaScript specification remain. Due to the limited time (±8 weeks) and all input scripts parsing successfully (that have been used to analyze the algorithms), the grammar has been considered good enough. I do not consider the impact of these possible deviations to be significant, but this could explain the difference between the output data of this paper and the previous research by Schafer et al. The considered deviation(s) that might occur, would impact the algorithms, as input could be incorrect or incomplete. This would affect the flow graphs that were used to extract call graphs from.

B. Interpretation

Several things have been unclear in the specification of the two algorithms by Schafer et al. An example is the flow graph creation rule for *function declarations* **R7**). The algorithm does not seem to distinguish global function declarations from in-scope ones (functions that are declared in the scope of another function). The rule specifies function declarations to be added to the flow graph in the following form:

$$Fun(\pi) \rightarrow Var(\pi)$$

The original specification considers a lookup function for local variables, which is mentioned as follows:

> We assume a lookup function λ for local variables such that $\lambda(\pi, x)$ for a position π and a name x returns the position of the local variable or parameter declaration (if any) that x binds to at position π.

In case a global function declaration would also be considered as a Var vertex, it would need to be added the symbol table, which does not comply the definition of the lookup function that only considers local variables. Ignoring this would result in not resolving global function declaration f through a transitive closure in both the algorithms. In case both the vertices in the function declaration and the function call would result in the same Vertex, the algorithm could determine where and which global function would be called, through the transitive closure. As a successive script, that would essentially implement the same algorithm (according to Schafer), considers globally declared functions as Prop vertices, I have chosen to implement this similarly. This decision could be a reason of the deviation between the results of this paper and those of the earlier paper by Schafer et al. It is unclear whether this same decision has been implemented the same way as in the successive script, because the aforementioned specification does not mention so.