Static JavaScript Call Graph Construction

Jorryt-Jan Dijkstra
Software Engineering
Centrum Wiskunde & Informatica / University of Amsterdam
jorryt-jan.dijkstra@student.uva.nl

I. Introduction

Call graphs are indispensable graphs for programming languages.

II. Grammar

Both call graph construction algorithms require a flow graph as input. This flow graph is constructed from a parse tree or abstract syntax tree. In order to get to such tree, a grammar had to be produced to parse JavaScript in the Rascal meta-programming language.

III. Flow Graph

The original call graph algorithms rely on abstract syntax trees (ASTs) as an input. These trees are processed to create flow graphs from. Flow graphs track intraprocedural and interprocedural flow. Both call graph algorithms rely on these flow graphs and both have a different approach of connecting functions to function calls, using the flow graphs information. The prerequisite JavaScript grammar has been used to parse existing JavaScript files to parse trees. The conversion of parse tree to abstract syntax trees has been skipped, because the parse trees contain the same relevant information as ASTs for both call graph algorithms.
This paragraph will explain what the required flow graph entails and how it is obtained using a parse tree as input.

A. *Intraprocedural flow*

Intraprocedural analysis deals with the statements or basic blocks of a single procedure and with transfers of control between them [**?**, p. 3]. This analysis will be done before the interprocedural analysis. Several rules originating from Schäfer et al. lead to an intraprocedural flow graph [**?**, p. 5]. Note that the nodes being visited are nodes in the parse tree, which are all unique locations within the parse tree.

	node at π	**edges added when visiting** π
R1	$l = r$	$V(r) \rightarrow V(l)$, $V(r) \rightarrow$ **Exp**(π)
R2	$l \mid\mid r$	$V(l) \rightarrow$ **Exp**(π), $V(r) \rightarrow$ **Exp**(π)
R3	$t\ ?\ l : r$	$V(l) \rightarrow$ **Exp**(π), $V(r) \rightarrow$ **Exp**(π)
R4	l && r	$V(r) \rightarrow$ **Exp**(π)
R5	$\{f \colon e\}$	$V(e_i) \rightarrow$ **Prop**(f_i)
R6	function expression	**Fun**$(\pi) \rightarrow$ **Exp**(π)
		if it has a name: **Fun**$(\pi) \rightarrow$ **Var**(π)
R7	function declaration	if it is in scope: **Fun**$(\pi) \rightarrow$ **Var**(π)
		if it is global: **Fun**$(\pi) \rightarrow$ **Prop**(π)

Fig. 1. Rules for creating intraprocedural edges for the flow graph, based on parse tree nodes

A different edge is added for function declarations in global scope (R7), in order to handle function declarations in global scope. This is not documented by Schäfer et al., but it is assumed that they had this intraprocedural as well. This edge is necessary in order to provide a more complete call graph, which will be explained more in depth in section X.

The basic set of intraprocedural vertices consists of four different types:

$$V \;::=\; \textbf{Exp}(\pi) \quad \text{expression at position } \pi$$

The V function as referred to in figure 1, represents a function that maps expressions to corresponding flow graph vertices:

$$V(t^{\pi}) = \begin{cases} \textbf{Var}(\pi') & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = \pi' \\ \textbf{Prop}(\pi) & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = undefined \\ \textbf{Prop}(f) & \text{if } t \equiv e.f \\ \textbf{Exp}(\pi) & \text{otherwise} \end{cases}$$

The mapping function V is dependent on a symbol map. This symbol map stores which variables are declared at which position in the local scope.

B. *Scoping*

Scoping is important for the mapping function V which uses $\lambda(\pi, x)$ to find local variable x for position π. An accurate variable lookup implies a more accurate flow graph.
Scope in a programming language controls the visibility and lifetimes of variables and parameters [**?**, p.36]. JavaScript has two types of scoping. It has a *global* scope, which means that a global variable is defined everywhere in your JavaScript code. The second type of scoping in JavaScript is limited to the scope of a function. Variables declared within a functions are only defined within a functions body. These are called local variables and also entail the function its parameters [**?**, p.53]. Therefore this scope is known as *local* scope. Even though JavaScript does support block syntax, it does not contain block scoping [**?**, p.36]. For creating the flow graph these two types of scopes have been taken into account. Because the lookup function for the flow graph algorithm solely looks up local variables, global variables are not stored in the symbol map. These will always be resolved to *Prop* vertices by function V.

1) Hoisting:

2) Overriding:

IV. Call Graph Forms

V. Data

VI. 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:

$$Fun(\pi) \to Var(\pi)$$

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.

References

[1] T. Marlowe and B. Ryder, "Properties of data flow frameworks," *Acta Informatica*, vol. 28, no. 2, pp. 121–163, 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01237234

[2] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 752–761. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486887

[3] D. Crockford, *JavaScript: The Good Parts. Unearthing the Excellence in JavaScript.* O'Reilly, 2008.

[4] D. Flanagan, *JavaScript: The Definitive Guide*, 6th ed., ser. Definitive Guide Series. O'Reilly Media, Incorporated, 2011. [Online]. Available: http://books.google.nl/books?id=4RChxt67lvwC