Static JavaScript Call Graph Construction

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

PREFACE

Thank you coffee machine.

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. This is due to Rascal being the programming language being developed at CWI. The language is used to carry out research. Therefore I invested approximately two months on creating a JavaScript grammar. The implementation of the grammar was led by the ECMAScript 5.1 (ECMA-262) specification of June 2011 [**?**]. In this chapter, some of the common problems in JavaScript will be discussed.

A. Automatic SemiColon Insertion

JavaScript does not require semicolon's after each written statement. Semicolon's are one of the ways to mark separate statements. Line terminating characters are treated as a semicolon, in case the next non space character after the line break can not be interpreted as a continuation of the current statement [**?**, p. 25]. This is called *automatic semicolon insertion*. The following characters are considered line terminating characters: *carriage return*, *line feed*, *line separator*, *paragraph separator* [**?**, p. 15]. The following sample code would result in declaring variable x with 7 due to automatic semicolon insertion:

```
1 var x = 1+
2 2
3 *3
```

A closing curly brace could also serve as terminating character for statements, provided that the code code is correct. Thus the brace should be closing a block.
Automatic semicolon insertion applies differently for *return*, *continue*, *break*, *throw* statements [**?**, p. 26] [**?**, p. 26]. A line terminating character after any of these keywords will be assumed to be the end of the statement. In the following example, the function invocation on line #3 in the following code can be considered dead code:

```
1 function f() {
2     return
3     g(10);
4 }
```

Furthermore postfix operators require to be on the same line as the its operand [**?**, p. 27]. Therefore ++ and – operators on a separate line, are not considered postfix operators for its previous connected operand, but prefix operators for its connected operand ahead. If JavaScript can not be parsed, the parser will insert a semicolon at the end of the input stream [**?**, p. 27].
The grammar that has been developed as a prerequisite of this research, takes automatic semicolon insertion into account for closing braces and linefeeds. It also recognizes the four exceptional statements with the alternative semicolon insertion.

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**(π) $\rightarrow$ **Exp**(π)
		if it has a name: **Fun**(π) $\rightarrow$ **Var**(π)
R7	function declaration	if it is in scope: **Fun**(π) $\rightarrow$ **Var**(π)
		if it is global: **Fun**(π) $\rightarrow$ **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 edge in their specification 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:

$$
\begin{array}{lll}
V & ::= & \textbf{Exp}(\pi) \text{ expression at position } \pi \\
 & \mid & \textbf{Var}(\pi) \text{ variable declared at position } \pi \\
 & \mid & \textbf{Prop}(f) \text{ property of name } f \\
 & \mid & \textbf{Fun}(\pi) \text{ function declaration/expression at } \pi
\end{array}
$$

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: An important feature of the JavaScript scope is called *hoisting*. This feature implies that variable declarations are hoisted to the top of the corresponding scope [**?**, p. 54]. Therefore declaring a variable anywhere in a scope, is equal to declaring it at the top of the scope [**?**]. Variables that are not yet initialized but are hoisted, will be considered undefined until they are initialized. Hoisting has been taken into account in order to refer to the correct position with mapping function V in the flow graphs. Due to hoisting support the reference to variable x on line #3 will refer to the declaration on line #4 rather than line #1:

```
1 var x = 3;
2 function f() {
3     console.log(x);
4     var x = 20;
5 }
```

2) Overriding: Global variables are defined through the program, whereas local variables are defined through its enclosing function and nested functions [**?**, p. 55]. Variables on the scope can be redeclared within functions in the nested scope, whereas redeclaration of a variable in the same scope is ignored in JavaScript. This can also be done by providing a parameter with the same name, so that the previous variable is overridden within the nested scope.
Function names are also variables, and can be overridden like any other variable, within nested scopes. In addition to this rule, a function name on the scope can be overridden within its own parameters. The following example of function f clarifies this:

```
1 (function f(f) {
2     return f;
3 })(10);
```

The sample code will return 10, due to the overriding of function name f within its scope.
Overriding of variables has been adopted to the flow graph creation algorithm, so that mapping function V will benefit from the most accurate variable references.

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:

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

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.

REFERENCES

[1] 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

[2] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[3] 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

[4] 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

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