Visual Facts JS: An Extended Call-Graph Generator Library For Javascript

Nigel Lee

School of Computer Science and Engineering

Nanyang Technological University

August 2, 2021

Submitted in Partial Fulfillment of the Requirements for the Degree of Engineering Science (Computer Science) of the Nanyang Technological University

This page is intentionally left blank.

Abstract

Hello

Contents

Chapter 1

Introduction

1.1 Background

1.2 Dynamic Languages

Dynamic languages can be generalised as languages that differ several behaviours of the language to runtime, that would otherwise occur during compile time for static languages.[1]

For the purpose of this project, we will focus on dynamic languages in the context of Javascript. In particular, Javascript exhibits the following behaviours which contribute to its dynamic nature.

1.2.1 Eval

In Javascript, the *eval* function takes a string as parameter and evaluates it as if it were Javascript source code, returning the result of evaluating the string.[2]

```
1  eval("let a = 1; a + 1;"); // 2
```

In the example above, the argument string passed to *eval* is interpreted as valid Javascript code and evaluates to 2.

This mechanism allows code to be changed at runtime, making Javascript a dynamic language. However, there has been considerable criticism against the usage of *eval*. Most notably, *eval* is unsafe[3] and prevents an optimising compiler

[1] https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

[3] https://developers.google.com/web/fundamentals/security/csp#eval_too

from conducting data flow optimisations[4].

1.2.2 Object Modification

Objects in Javascript can be modified at runtime by adding or removing property fields and behaviour. Contrary to a more static language like Java where object instances are created based on the attributes and behaviour of the instantiating class, Javascript objects can add properties via the *Object.defineProperty()*[5] and delete properties via the *delete*[6] operators.

Functions can be defined on objects at runtime in the following manner:

```
1 let obj = {};
2 Object.defineProperty(obj, "fooFunc", function () {
3   return 1;
4 });
```

In the above example, *fooFunc* is only attached to *obj* at runtime. It would not be possible to conduct static analysis on the location of *fooFunc* using the information available at compile time.

1.2.3 Dynamic Types

Types in Javascript are determined at runtime and can be altered at runtime. An AST[7] generated at compile time does not capture any information about the type of the variable.

In the following example, an AST is generated for a variable declaration. It can be observed that no information is available of the type of variable *a*. Its type can be determined using the *typeof* operator at runtime only.

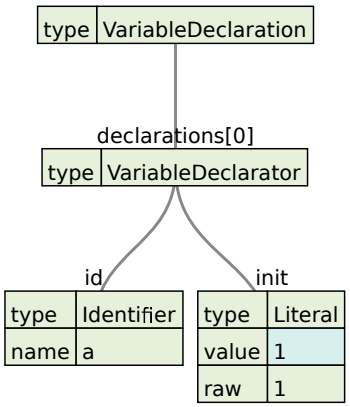
```
1 let a = 1;
2 typeof a; // number
```

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#never_use_eval!

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete

[7] Abstract Syntax Tree



In addition, the type of *a* can be altered at runtime. It is possible to assign a string or function to *a*, changing its type.

```
let a = 1;
typeof a; // number

a = "1";
typeof a; // string
```

Thus, it would be impossible to conduct static analysis by relying on the type system of a dynamic language.

1.3 Challenges

In addition to the challenges posed by the dynamic features of Javascript, there are several other features that increase the complexity of conducting static analysis on Javascript code.

1.3.1 First Class Functions

Functions in Javascript are treated as *first class citizens*[8], similar to other primitive types such as *number* and *string*. Function expressions support the following behaviour:

- Assignment to variables.
- Passed as arguments into functions. This is commonly referred to as a *callback*.
- Returning a function definition from an outer function. This is commonly used in *Higher Order Functions*[9] where an outer function (considered the

[8] https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function

[9] https://www.oreilly.com/library/view/functional-javascript/9781449360757/ch04.html

higher order) returns an inner function to be invoked later on.

In order to account for this behaviour of functions, we need to conduct data flow analysis to locate the position of functions.

1.3.2 Dynamic Object Access

Objects properties in Javascript can be accessed using the dot *"."* notation or braces *"[ ]"* notation. The difference between the two is that the dot notation requires that a valid identifier[10] be used directly while the braces notation will first try to evaluate the expression within the braces to yield a computed identifier.

```
1  const obj = {
2    a: function () {
3      return 1;
4    },
5  };
6
7  function getBChar() {
8    return "a";
9  }
10
11 obj.a(); // 1;
12
13 obj["a"](); // 1;
14
15 obj[getBChar()](); // 1;
```

In the example above, *obj* has a member function which returns 1. All three expressions on Lines 11, 13, 15 are valid ways of accessing the member function *a*.

There is no feasible way to determine the return value of *getBChar* at compile time without resorting to heuristics. As such, we will ignore dynamic object access using braces for this project.

1.4 Extended Call Graphs

In order to conduct static analysis on Javascript code, we will need to utilise call graphs. We will extend the definition of call graphs to include variable relations.

Call graphs provide a visualisation of the relations between functions in a codebase. When a function *foo* calls another function *bar* within its body, we can describe this process as *foo* invoking *bar*.

[10] `https://developer.mozilla.org/en-US/docs/Glossary/Identifier`

```
function foo() {
  bar();
}
```

In a complex codebase with a large number of invocations, a graph representing these invocations would be useful in helping a developer to quickly understand the codebase.
Call graphs with function - function relations are defined as follows:

Definition 1. *A call graph $G(V, E)$ consists of the vertex set V representing functions and edge set E representing invocations such that $e_{ij} \in E$ is a directed edge from $v_i \in V$ to $v_j \in V$ if function i invokes function j.*

It would be helpful for the developer if a similar relation can be used to visualise when functions refer to variables outside its scope.

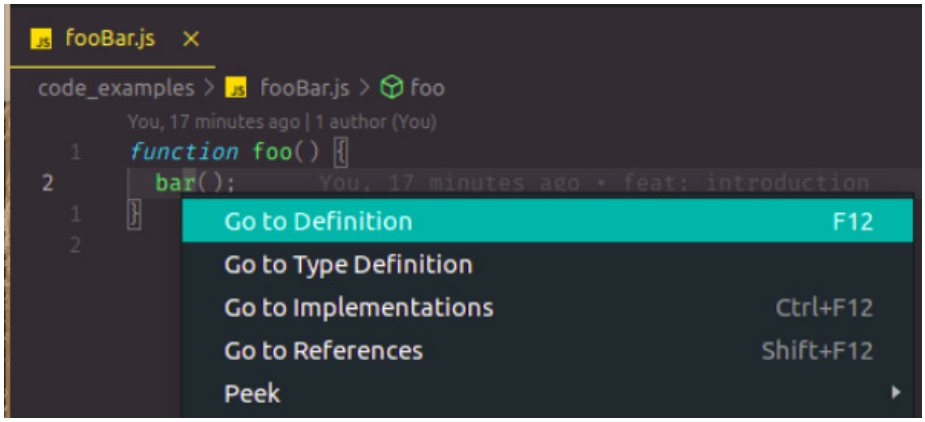
```
let a = 1;
function foo() {
  return a + 1;
}
```

This project attempts to extend the call graph by adding function - variable relations, resulting in the following definition:

Definition 2. *An extended call graph $G'(V', E')$ consists of the vertex set V' representing functions and variables, and the edge set E' representing invocations and references such that $e_{ij} \in E'$ is a directed edge from $v_i \in V'$ to $v_j \in V'$ if function i invokes function j or if function i references variable j.*

1.5 Applications

Extended call graphs can be used internally to power *go to definition* features in IDEs[11].



[11] Integrated Development Environment

Looking up the definition of a function or variable can be simplified to the straightforward task of traversing the extended call graph to find the source and destination vertices.

Another application for extended call graphs is to generate UML-like[12] diagrams for dynamic languages to visualise the relations in a codebase and improve developer productivity.

1.6 Prior Work

There were several projects that have attempted to generate call graphs using approximate methods. Most notably, *Schaefer et al*[13] devised a technique which accounts for First Class Function behaviour using data flow analysis.

This project builds upon the work done by *Schaefer et al* by including function - variable relations in an extended call graph.

[12] `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/`

[13] TODO

Chapter 2

Implementation

2.1 Parsing

2.2 Module Overview