JSFox: Integrating Static and Dynamic Type Analysis of JavaScript Programs

Tian Huat Tan[1], Yinxing Xue[2], Manman Chen[3], Shuang Liu[4], Yi Yu[5], and Jun Sun[3]

[1] Acronis, Singapore
[2] Nanyang Technological University
[3] Singapore University of Technology and Design
[4] Singapore Institute of Technology
[5] National Institute of Informatics, Japan

Abstract. JavaScript is a dynamic programming language that has been widely used nowadays. The dynamism has become a hindrance of type analysis for JavaScript. Existing works use either static or dynamic type analysis to infer variable types for JavaScript. Static type analysis of JavaScript is difficult since it is hard to predict the behavior of the language without execution. Dynamic type analysis is usually incomplete as it might not cover all paths of a JavaScript program. In this work, we propose JSFox, a browser-agnostic approach that provides integrated type analysis, based on both static and dynamic type analysis, which enables us to gain the merits of both types of analysis. We have made use of the integrated type analysis for finding type issues that could potentially lead to erroneous results. JSFox discovers 23 type issues in existing benchmark suites and real-world Web applications.

1 Introduction

JavaScript is arguably one of the most used programming languages [4, 6]. It has been supported by all modern Web browsers. It can be executed on almost all kinds of platforms (mobile, PC, tablets, etc.) with various operating systems (Windows, Mac OS, Linux, etc.). Since the advent of Web 2.0, many functionalities which were originally implemented on the server, have migrated to the client-side with the help of JavaScript. Nowadays, many complicated Web applications, such as Gmail or Google Docs, are written entirely in JavaScript.

JavaScript is a dynamic, weakly typed language with many flexible features such as dynamic code evaluation, function variadicity, and constructor polymorphism. This can ease the programmer's job for writing compact code. Unfortunately, the freedom and dynamism of Javascript is a double-edged sword. No compile-time warnings are shown when variables with inconsistent types are used. Even worse, some values are coerced into another type, leading to incorrect behavior without any obvious sign of misbehavior. Type analysis for JavaScript can help mitigate these issues and improve the development efficiency. Type analysis is crucial for capturing representation errors, e.g., misusing a number as

	(a)	(b)
Program	3D-RayTrace (SunSpider)	AES Encyption (Web Application)
Source code	1 `function invertMatrix (self) {` 2 `  var temp = new Array(16);` 3 `  var tx = -self[3];` 4 `  ...` 5 `  self[3] = tx * self[0] + ...;` 6 `  ...` 7 `  return self;` 8 `}` 9 `...` 10 `var m = new Array(16);` 11 `m[0] = xaxis[0]; ...` 12 `m[4] = yaxis[0]; ...` 13 `m[8] = zaxis[0]; ...` 14 `invertMatrix(m);`	1 `setKey: function(key)` 2 `{` 3 `  var key = this._blockGen(pad(key),true);` 4 `  ...` 5 `}` 6 `...` 7 `var k = decode("f..", 2);` 8 `aes.setKey(k);`
Potential Type Issues	Variable `tx` has type `NaN`, and `self[3]` has types `Undefined` and `NaN`	Variable `key` has types `string` and `array object`
Problem	`m` behaves differently at different indices	`key` behaves differently at different locations

Fig. 1: Real-world problems related to potential type issues

an array. Moreover, type information is the basis for program analysis methods like symbolic execution [22], and can serve as an abstraction for analysis methods like testing and model checking [11].

Existing works use either static [9, 23, 30] or dynamic [26] analysis to infer variable types in JavaScript programs. However, both approaches have their limitations. Static type analysis is often restricted to a subset of JavaScript language features in order to achieve soundness. Some information, e.g., values returned by external calls, can not be reached by pure static analysis. To perform analysis for the entire JavaScript program, unsound static type analysis has been adopted (e.g., [24]). Even so, many dynamic features of JavaScript remain very difficult to infer statically. Dynamic type analysis can address the challenges caused by the dynamic features of JavaScript. Nevertheless, unlike static type analysis, dynamic type analysis is mostly incomplete, as dynamic execution might not cover all program paths in the JavaScript program.

We demonstrate the complication in type analysis of JavaScript programs using two previously unreported potential type issues in Figure 1. Figure 1(a) is a JavaScript program from a popular JavaScript benchmark [5]. At line 10, `m` is created as an array of 16 elements and different elements in the array are assigned with different numbers from lines 11 to 13. Nevertheless, some elements in the array `m`, e.g., `m[3]`, are not assigned with any value, therefore, it is set to

`undefined` by default. Subsequently, `invertMatrix` is invoked, with the array `m` as input. As a consequence, `self[3]` is set to be `undefined`. `tx` is then set to the negation of `undefined`, which results in a special value in JavaScript, `NaN`, signaling an error. Subsequently, setting `tx` to be `NaN` results in setting `self[3]` to be `NaN` (line 5). As a result, the array `m` contains `NaN` at certain indices, while numbers in other indices. Pure static type analysis is hard to analyze the value for `self[3]` since `self` is passed as a function parameter. Although pure dynamic type analysis can deduce a value for `self[3]` on one program path, it is in general hard to reason under multiple program paths.

Figure 1(b) is a real-world JavaScript program from [2]. At line 7, the function is invoked with the variable `key` which is of type string. At line 3, a variable `key` is redeclared with a type of array object. In JavaScript, a type is either local or global to a function. Since function parameter `key` and the redeclared variable `key` are declared in the same function scope, they are resolved to the *same* variable. For example, if the function parameter `key` at line 1 has value `"hello"`, and is then changed to `var key;` (i.e., declaring variable `key`) at line 3, then variable `key` at line 3 will still contain the value `"hello"`. Therefore, it means that, variable `key` which is of type `string` is reassigned with an array object. Since array objects and string have different behaviors, variable `key` behaves differently at different locations. Similarly, pure static or dynamic type analysis is hard to reason about the values for `key` in a precise and complete manner.

The two examples above highlight the difficulties in type analysis using static or dynamic analysis alone. In this work, we propose JSFox, a framework of integrated type analysis that bridges the gap between static and dynamic type analysis, and combines the strength of both approaches. JSFox is browser-agnostic, and targets entire JavaScript syntax. We illustrate briefly how JSFox performs type detection via integrated type analysis. Firstly, JSFox leverages static type analysis to infer the type information of each variable in JavaScript programs. To enable analysis for all JavaScript syntax rather than a subset of it, an unsound static type analysis is adopted. Secondly, JSFox performs dynamic type analysis to obtain the types of variables and call-graph information of the JavaScript program. Lastly, the integrated type analysis is done by integrating the results from static and dynamic type analysis.

In short, we present a novel approach that combines static and dynamic analysis to infer types for JavaScript programs, which could be used for various purposes such as detecting potential type issues in JavaScript programs. JSFox has been evaluated on several real-world Web applications and a popular JavaScript benchmarks collection, i.e., JetStream [3], which includes benchmarks from the SunSpider 1.0.2 [5] and Octane 2 [13]. JSFox has been shown to be effective in identifying type issues so that developers can easily fix the problem and improve the code. There are 23 potential type issues reported in existing benchmark suites and real-world Web applications.

We summarize the contributions in the following.

```
1  var y=0;
2  var x="a";
3  var i=1;
4  function f(a){
5    return a;
6  }
7  function chkFlag(flag){
8     var code=arguments[i];
9     if(flag==0){
10      return x*1;
11    }else if(flag==1){
12      return 1/y;
13    }else if(flag==2){
14      return -1/y;
15    }else{
16      return eval(code);
17    }
18 }
19 var res=chkFlag(3,"f('res')");
```

Fig. 2: A JavaScript example program

- We propose an integrated type analysis for JavaScript programs which facilitate the usage of dynamic type analysis in refining static type analysis, for the purpose of inferring types and finding type issues.
- We have developed JSFox, which is browser-agnostic and targeted at all features of Javascript.
- We have evaluated our approach on popular JavaScript benchmarks, and several real-world Web applications. The evaluation shows that our method has detected 23 potential type issues with the low false positive rate.

Outline. The rest of this paper is structured as follows. Section 2 provides a motivating example and defines the problem to be solved in this work. Section 3 introduces the architecture of JSFox and various components in JSFox. Section 4 introduces the integrated type analysis of JSFox. Section 5 presents our implementation and evaluates our approach. Section 6 reviews related works. Finally, Section 7 concludes the paper, and outlines future work.

2 Overview and Example

In this section, we present a motivating example and then define the problem to be solved.

2.1 Motivating Examples

Figure 2 shows a JavaScript program, which starts by invoking the function `chkFlag` at line 19. Note that two parameters are passed when function chkFlag is invoked. However, only one argument is declared in the function definition at line 7. This is allowed by JavaScript, since it supports *variadic functions*. A function in JavaScript can be invoked with any number of arguments, regardless of the function definition. If fewer arguments are given than that in the definition,

the values for the remaining declared arguments are set to be `undefined`. If more arguments are given than that in the definition, those additional arguments can be obtained through a JavaScript built-in variable called `arguments`. In our case, the second argument is obtained through the `arguments` at line 8.

Function `chkFlag` returns different results based on the value of `flag`. At line 10, it returns `NaN` (Not A Number), since `"a"*1` cannot be evaluated to a number. At lines 12 and 14, the function returns `Infinity` and `-Infinity` respectively, since denumerators `y` are zero in both cases. Note that `NaN`, `Infinity`, and `-Infinity` are all special values in JavaScript that signify issues in the evaluation.

At line 16, function `eval` is called to evaluate the program contained in variable `code` at runtime. Variable `code` contains `"f('res')"`, which represents the invocation of function `f` with an argument `'res'`. Eventually, the value `'res'` is returned at line 5, which is subsequently returned at line 16. Therefore, variable `res` at line 19 is assigned with the value `'res'`.

This motivating example contains a variadic function, and the `eval` construct for evaluating dynamically loaded code. These dynamic features contribute to the dynamism of JavaScript, and cannot be precisely modeled in static type analysis. For example, `arguments[i]` at line 8 can only be known at runtime. In addition, covering lines 10, 12, 14 might not be easy in dynamic type analysis, since these lines do not execute with `flag=3`. Although in our case, we can cover these lines by modifying the value of `flag`, it is very hard in general to explore different paths with dynamic type analysis. Therefore, we propose integrated type analysis in JavaScript to take the best of both worlds – using static type analysis for good coverage and dynamic type analysis for resolving dynamic type features.

2.2 Types in JavaScript

We start with briefly introducing the type paradigm in JavaScript. There are two categories of types in JavaScript – primitive and object types. Primitive types include boolean, number, string, undefined, and null. The object type encapsulates several properties in their own types. In JavaScript, a function is treated as an object, and it can be passed as an argument to a function or assigned to a variable. In our work, we classify the types in JavaScript into seven domains – undefined (`Undef`), null (`Null`), bool (`Bool`), number (`Num`), string (`String`), object (`AllocSite`), and error (`Err`). In this work, a domain is defined using a lattice, which is visually represented using a Hasse diagram. The Hasse diagrams of the seven domains are provided in Figure 3.

We introduce some of the design decisions of the Hasse diagrams. The bottom symbol, $\perp$, denotes no value. `Num` and `String` are refined to recognize unsigned integer (*ArrString* and *ArrInt* resp.). The usage of unsigned integer (from 0 to 2^{32}–1=4294967295) is mainly to detect the correctness of array indices, since only values from unsigned integer are allowed for array indices [25]. The `AllocSite` is used to record the start line number of the object definition. Consider the example in Figure 2, if we have a statement `var obj=chkFlag; obj` will have

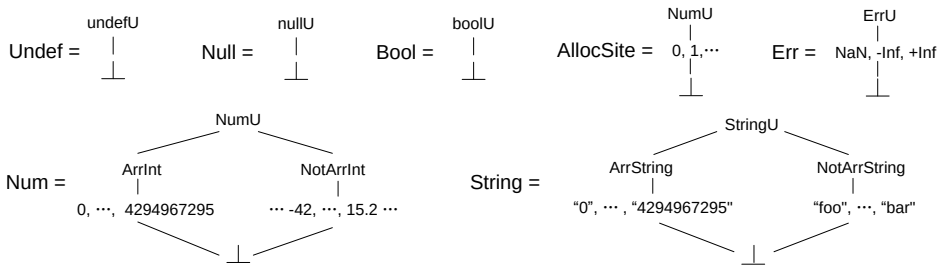

Fig. 3: Hasse diagrams

value 7. The `Err` index is used to accumulate information that signals potential problems. They include:

- **Not a number** (*NaN*): Not-A-Number (`NaN`) is derived by invalid operations on numbers, e.g., line 10 in Figure 2.
- **Positive Infinity** (*+Inf*): Positive Infinity (`Infinity`) is derived by 'dividing positive number by zero', e.g., line 12 in Figure 2.
- **Negative Infinity** (*-Inf*): Negative Infinity (`-Infinity`) is derived by 'dividing negative number by zero', e.g., line 14 in Figure 2.
- **Error in array indices** (*IdxErr*): The read of an array $a[i]$ may access an index i that is not an unsigned integer, e.g., a[-1].

Those potential problems are automatically classified as exceptions in languages such as Java, but they are allowed in JavaScript.

2.3 Type Configuration and Valuation

In this section, we define type configuration and valuation that will be used to define our problem of type analysis.

Definition 1 (Type Configuration). *A type configuration is defined as* $(T, \prec, \bigsqcup, H)$ *where*

$$T = \texttt{Undef} \times \texttt{Null} \times \texttt{Bool} \times \texttt{Num} \times \texttt{String} \times \texttt{AllocSite} \times \texttt{Err}$$

with $\prec = \{\prec_1, \cdots, \prec_7\}, \bigsqcup = \{\bigsqcup_1, \cdots, \bigsqcup_7\}$, *and* $H = \{H_1, \cdots, H_7\}$ *define the ordering operators* $\prec_i$, *join operators* $\bigsqcup_i$ *and domain cardinalities* H_i, *where* $i \in \mathbb{Z} \cap [1, 7]$.

$T[i]$ is used to denote the i-th component of T, e.g., $T[2]$ refers to the `Null` domain. We call $t \in T$ a *type*, and we use $t[i]$ to denote the i-th component of t, e.g., suppose $t = (\bot, nullU, \bot, ArrInt, \bot, \bot, \bot)$, we have $t[4] = ArrInt$. Sometimes, we also refer to the component in type t by name, e.g., $t[\texttt{Num}] = ArrInt$. For conciseness, we also use ($\texttt{Null} : nullU$, $\texttt{Num} : ArrInt$) to represent $(\bot, nullU, \bot, ArrInt, \bot, \bot, \bot)$, by only expressing types that have non-bottom values. In addition, we use $(\bot)$ to denote the type with all bottom values $(\bot, \bot, \bot, \bot, \bot, \bot, \bot)$.

The ordering operators $\prec_i$ are defined according to the Hasse diagrams in Figure 3 where $i \in \mathbb{Z} \cap [1, 7]$. For example, we have $ArrInt \prec_4 NumU$ according to the Hasse diagram for `Num`. In some lattices like `Num`, there are comma separated

values, e.g., the numbers from 0 to 429496725 under *ArrInt*. These denote power sets of these numbers, e.g., {1,2} or {1,2,3}. These sets are ordered using "is subset of" relation, e.g., $\{1,2\} \prec_4 \{1,2,3\} \prec_4 \mathit{ArrInt}$. We use $\prec$ to denote the ordering relation between types. Given two types $t_1, t_2 \in T$, $t_1 \prec t_2$ if for each $i \in \mathbb{Z} \cap [1,7]$, $t_1[i] \prec t_2[i]$, and we say that t_1 *is smaller than* t_2.

The join operators $\bigsqcup_i$ are defined according to the Hasse diagrams in Figure 3 where $i \in \mathbb{Z} \cap [1,7]$. For example, we have $\mathit{ArrInt} \bigsqcup_4 \mathit{NotArrInt} = \mathit{NumU}$. We use the join operator $\bigsqcup$ for joining two types $t_1, t_2 \in T$, e.g., given $t_1 = (\texttt{Num} : \mathit{ArrInt})$ and $t_2 = (\texttt{Num} : \mathit{NotArrInt})$, $t_1 \bigsqcup t_2 = (\texttt{Num} : \mathit{NumU})$.

Given a type $t \in T$, the domain cardinality, H_i, denotes the maximum number of values allowed in $t[i]$. For example, if $H_4 = 2$, then $\{1,2\}$ is allowed in `Num`, but $\{1,2,3\}$ is not allowed in `Num`. The join operator $\bigsqcup_i$ works according to H_i. For example, given $H_4 = 3$, we have $\{1,2\} \bigsqcup_4 \{3\} = \{1,2,3\}$; while given $H_4 = 2$, we have $\{1,2\} \bigsqcup_4 \{3\} = \mathit{ArrInt}$.

Definition 2 (Type Valuation). *Given a program P, the* type valuation $Q : \mathit{Var}(P) \mapsto T$ *is a (partial) function that maps a variable $v \in \mathit{Var}(P)$ to a type $t \in T$, where $\mathit{Var}(P)$ is the set of all variables in P.*

Assume we have a JavaScript program: `var a=1,b="hello";` Then, the type valuation of the program is $\{a \mapsto (\texttt{Num} : \{1\}), b \mapsto (\texttt{String} : \{\text{"}\mathit{hello}\text{"}\})\}$. In this case, we have $Q(a) = (\texttt{Num} : \{1\})$ and $Q(b) = (\texttt{String} : \{\text{"}\mathit{hello}\text{"}\})$.

2.4 Potential Type Issues

Programmers tend to make use of consistent types in programming to facilitate understanding. Certain potential type issues usually correlate with programming problems that developers need to pay attention to [26]. We divide the potential type issues into two categories: potential inconsistent types and potential exception types. A variable has a potential inconsistent type if it has been used for two different types. For example, a variable v with type valuation $Q(v) = (\texttt{Num} : \{1\}, \texttt{String} : \{\text{"}a\text{"}\})$ has a potential inconsistent type, as the same variable is assigned with values from both `Num` and `String` types. It is defined formally as follows.

Definition 3 (Potential Inconsistent Type). *A variable v is said to have an* potential inconsistent type *if the type valuation of v, $t = Q(v)$, satisfies the following condition:*

There exist a_i, a_j where $a_i, a_j \in \{\texttt{Bool}, \texttt{Num}, \texttt{String}, \texttt{AllocSite}\}$ and $a_i \neq a_j$, such that $t[a_i] \neq \bot$ and $t[a_j] \neq \bot$.

The values $e \in \texttt{Err}$ (e.g., `NaN`), are atypical to many strong-type languages like Java, since operations that generate these values normally result in exceptions in these languages. Therefore, the variables that could have values $e \in \texttt{Err}$ are said to have an *exception type*. A programmer needs to take care of them properly, and otherwise they can become a source of errors. We define the exception type formally as follows.

Definition 4 (Potential Exception Type). *A variable v is said to have* potential exception type *if the type valuation of v, $t = Q(v)$, has the condition $t[\texttt{Err}] \neq \bot$.*

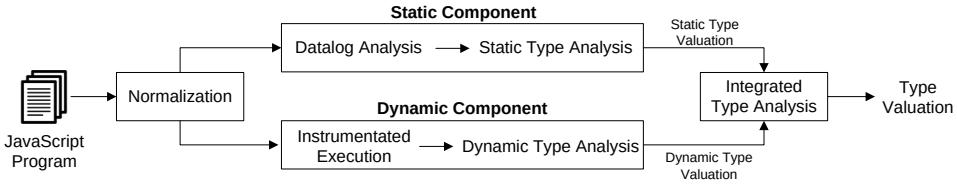

Fig. 4: Architecture of JSFox

```
1  (function (__global) {
2       var tmp0, tmp1,....;
3       tmp13 = function (a) {
4           var tmp14;
5           tmp14 = a;
6           return tmp14;
7       };

69      tmp7 = tmp16(tmp10, tmp11);
70      tmp6 = 'res';
71      __global[tmp6] = tmp7;
72  }(typeof global === 'undefined' ?
        this : global));
```

```
10      tmp16 = function (flag) {
11         ...
12         tmp20 = tmp21 == tmp22;
13         if (tmp20) {
14              tmp26 = 'x';
15              tmp24 = __global[tmp26
                    ];
16              tmp25 = 1;
17              tmp23 = tmp24 * tmp25;
18              return tmp23;
19          }
```

Fig. 5: Normalized JavaScript example

Henceforth, we simply refer respective terms without the word "potential" for simplicity (e.g., potential type issues as type issues). In Section 3, we introduce JSFox. The goal of JSFox is to detect type issues using integrated type analysis.

3 Architecture of JSFox

The architecture of JSFox is shown in Figure 4. The input of JSFox is a JavaScript program and the output would be the type valuation of the JavaScript program. The input JavaScript program is first normalized. The normalized program is then analyzed using static and dynamic components of JSFox respectively. The output of these components will be static and dynamic type valuations. Both kinds of type valuations are then integrated by integrated type analysis to yield the final type valuation.

Static component is composed of two steps: Datalog analysis and then static type analysis. For dynamic component, there are also two steps: instrumented execution and dynamic type analysis. In this section, we discuss Datalog analysis and instrumented execution. The remaining steps are introduced in Section 4.

3.1 Normalization

A JavaScript program is first normalized into a three-address-code-like [8] format that is amenable for analysis. For example, our Datalog analysis requires a normalized JavaScript program as input. Each JavaScript statement is normalized such that there is at most an operator on the right hand side (RHS) of the statement. For example, given a statement `d=a*b+c` that contains two operators on RHS of the statement, the statement is normalized into two sequential statements, i.e., `tmp1=a*b; d=tmp1+c;` where `tmp1` is a temporary variable. In addition, all variables are alpha renamed during the normalization. Therefore, all variables

have distinct names after the normalization. We modify JS_WALA [10] with alpha renaming for the purpose of normalization, and the complete syntax for normalized JavaScript programs can be found in [10]. The normalized program of the example in Figure 2 is shown in Figure 5. The mapping of statements from Figure 2 to Figure 5 should be self-explanatory.

```
[AllocRule]
VarPointsTo(variable, heap) :-
Reachable(methHeap), Alloc(
    variable, heap, methHeap)

[FunctionCallRule]
Reachable(toMethHeap),
    CallGraph(invo, toMethHeap)
      :-
VCall(variable, invo,
    inMethHeap), Reachable(
    inMethHeap),
VarPointsTo(variable,
    toMethHeap)
```

Fig. 6: Datalog rules

```
[AllocRule]
VarPointsTo(tmp16, 10):-
Reachable(1), Alloc(
    tmp16, 10, 1)

[FunctionCallRule]
Reachable(10), CallGraph
    (69, 10):-
VCall(tmp16, 69, 1),
    Reachable(1),
VarPointsTo(tmp16,10)
```

Fig. 7: Inference results for motivation example

3.2 Datalog Analysis

Datalog analysis includes control flow analysis and pointer analysis. An important part of control flow analysis is call-graph discovery. Call-graph discovery provides information on which function definition is invoked by a function application, and pointer analysis provides the information on which object a variable is pointed to. The call-graph discovery and pointer analysis are performed simultaneously since they are mutually independent. We adopt an approach similar to [24] for call-graph discovery and pointer analysis using Datalog. The analysis is field-sensitive, that is, it distinguishes properties of different objects. Different from [24], our rules also include the analysis of reachability of a function, which can be used to reduce the search space of Datalog. A set of facts are collected from the normalized JavaScript program. Subsequently, the collected facts, together with a set of Datalog rules are solved using Microsoft Z3 fixpoint solver [12] for pointer analysis and call-graph discovery.

There are a total of nine Datalog rules in our analysis, which can be found at [1]. Two of the Datalog rules for demonstrating purpose are shown in Figure 6. A rule consists of two portions separated by the :- symbol. The portion before the symbol is called the *head* of the rule, while the portion after the symbol is called the *tail* of rule. The head and tail of a rule are predicates separated by commas. Whenever all predicates in the tail hold, it infers that all predicates in the head must hold.

For `AllocRule`, it is concerned with object assignment. The predicate `Alloc(variable, heap, methHeap)` means that the parameter `variable` is assigned with function object that started at line `heap`, and the assignment is performed within the function started at line `methHeap`. We call the predicate with parameters replaced

with constant values as a *fact*, e.g., `Alloc(tmp16,10, 1)`. Figure 7 represents the application of the rules in Figure 6 for the example shown in Figure 5. For `FunctionCallRule`, it is concerned with function invocation.

Given a program P, the discovered call-graph will be used to construct the control flow graph, denoted by *C(P)*, of program P. Subsequently, we group the straight-line code sequence with no branches and under the same function as a *basic block*. The collection of all basic blocks for a program P is denoted as *B(P)*.

The basic blocks of example in Figure 2 are shown in Figure 8. Each rectangle in Figure 8 represents a basic block, which is labelled with a block name b_i, followed by the line numbers of statements that contained within the block. For example, "b_1:1,2,3,4,7" means that statements at lines 1,2,3,4 and 7 in Figure 2 are included in basic block b_1. The arrows in Figure 8 denote the control flow of different basic blocks and the solid edges refer to the control flow sequence that is inferred by control flow analysis in this section. Given a basic block $b \in B(P)$, we use *b.backwardBlocks* and *b.forwaredBlocks* to denote the basic blocks that b is connected from and connected to respectively. For example, for basic block b_{11}, $b_{11}.backwardBlocks = \{b_7, b_8, b_9\}$, and $b_{11}.forwardBlocks = \emptyset$ in Figure 8.

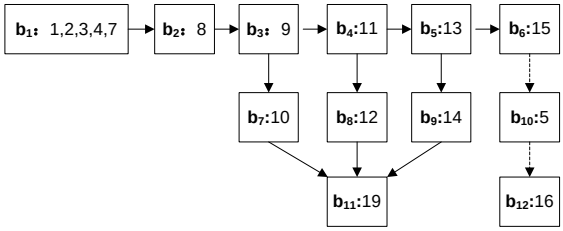

Fig. 8: Control flow with basic blocks

3.3 Instrumented Execution

We instrument the normalized JavaScript program in order to obtain the variable values and dynamic call graph edges. For collection of variable values, we record the values of variables that have been assigned at a particular line. For example, we record the value of `res` at line 19 in Figure 2, where the value is returned by function `chkFlag`.

For collection of dynamic call graph edges, we record the function definition that is invoked by a function application. We provide two examples for illustration. For the first example, given `chkFlag` is invoked at line 19 in Figure 2, we trace on which function definition is invoked. In this case, it is the function definition at line 7. The second example is that the application of `eval` function at line 16, will invoke line 4 subsequently. The call graph edges for the second example are represented using dotted edges in Figure 8. We remark that these are the edges which have been missed during Datalog analysis. After instrumenting the JavaScript program, we execute the instrumented program and collect the variable values and dynamic call graph edges. This information will be used for dynamic and integrated type analysis.

4 Integrated Analysis of JSFOX

In the following, we explain how static and dynamic type analysis are done first and then how they are integrated in JSFox.

4.1 Static Type Analysis

Our static analysis is field-sensitive, flow-sensitive, context-insensitive, and path-insensitive. We explain our design decision. Object properties are used frequently in JavaScript programs. Field-sensitive analysis enables us to distinguish between properties of an object. Thereby, it enables us to infer types for each property of an object. In order to handle large applications, we adopt context-insensitive and path-insensitive analysis. Our static type analysis is based on the monotone framework [21], and the algorithm for static type analysis is shown in Algorithm 1. The inputs of Algorithm 1 are a JavaScript program P, initial static type valuation, Q_{si}, and the output of the algorithm is the static type valuation Q_s.

Algorithm 1 iterates through multiple rounds (lines 2–12) for updating the type valuation of the program until a fixpoint is reached. In each round, the static type valuation Q_s is initialized to $\emptyset$ (line 1) and variable *changes* is initialized to $false$ (line 3). Subsequently, each basic block b of the program P, is iterated (line 4). The input type valuation of basic block b ($In[b]$) is the join of all output type valuations of backward blocks of b (line 6 and line 7). At line 8, the *update* function takes $In[b]$, b, and Q_{si} as inputs, updates the value of $In[b]$ using b and Q_{si}, which results in the output type valuation $Out[b]$. For example, consider $In[b] = \{i \mapsto (\texttt{Num} : \{1\}), j \mapsto (\texttt{String} : \{\text{"}hello\text{"}\})\}$, block b with two statements `i=i+1;j=obj;`, and $Q_{si} = \{obj \mapsto \texttt{AllocSite} : \{10\}\}$. Then, $Update(In[b], b, Q_{si})$=$\{i \mapsto (\texttt{Num} : \{1, 2\}), j \mapsto (\texttt{String} : \{\text{"}hello\text{"}\}, \texttt{AllocSite} : \{10\})\}$.

At line 9, *chg* is set to a boolean value that indicates whether $In[b]$ is the same with $Out[b]$. If *chg* is set to be true for any block b, it means that there exists a change in the current round, and *changes* is subsequently set to be true for the current round (line 11). The loop continues until it reaches a fixpoint where no block has changed (i.e., $changes = false$) (line 12). Subsequently, the static type valuation Q_s of the program P is obtained by including output type valuations of all basic blocks (line 14).

Although soundness is a desirable property for static analysis, it is very difficult to ensure soundness and yet to provide usable analysis. This is because JavaScript is a complex language with a number of dynamic features that are hard or impossible to statically analyze [28, 19]. In our static analysis, we adopt an unsound approach by giving the bottom value, i.e., ($\perp$) for the unknown. For example, line 16 in Figure 2 returns the bottom value since $eval(code)$ is unknown without executing the program.

Algorithm 1: Static Type Analysis

input : JavaScript program, P
input : Dynamic call graph edges, E_d
input : Initial static type valuation, Q_{si}
output: Static type valuation, Q_s

1 $Q_s \leftarrow \emptyset$;
2 **do**
3 $\quad$ *changes* $\leftarrow$ *false* ;
4 $\quad$ **for** $b \in B(P)$ **do**
5 $\quad\quad$ $In[b] \leftarrow (\bot)$;
6 $\quad\quad$ **for** $b_i \in b.backwardBlocks$ **do**
7 $\quad\quad\quad$ $In[b] \leftarrow In[b] \bigsqcup Out[b_i]$;
8 $\quad\quad$ $Out[b] \leftarrow Update(In[b], b, Q_{si})$;
9 $\quad\quad$ $chg \leftarrow (In[b] \neq Out[b])$;
10 $\quad\quad$ **if** *chg* **then**
11 $\quad\quad\quad$ *changes* $\leftarrow$ *true*;
12 **while** (*changes*);
13 **for** $b \in B(P)$ **do**
14 $\quad$ $Q_s \leftarrow Q_s \cup Out[b]$;
15 **return** Q_s ;

Algorithm 2: Dynamic Type Analysis

input : JavaScript program, P
input : Collected values for variables, V_d
output: Dynamic type valuation, Q_d

1 **for** $v \in Var(P)$ **do**
2 $\quad$ $Q_d(v) \leftarrow (\bot)$;
3 $\quad$ **for** $q \in V_d(v)$ **do**
4 $\quad\quad$ $D \leftarrow getDomain(q)$;
5 $\quad\quad$ $Q_d(v)[D] \leftarrow Q_d(v)[D] \bigsqcup q$;
6 **return** Q_d ;

4.2 Dynamic Type Analysis

The instrumented JavaScript program is executed during dynamic analysis. All variables and their collected values are represented using V_d. V_d is a function that maps a variable to its collected values. For example, given a variable v, $V_d(v) = \{1, "abc"\}$ means that the variable v is observed to have two values $\{1, "abc"\}$ during executions.

The main objective of dynamic type analysis is to transform the values of variables collected from execution of instrumented JavaScript (V_d) into dynamic type valuation Q_d. The algorithm for dynamic type analysis is in Algorithm 2. The inputs of Algorithm 2 are the JavaScript program P and the collected values for variables, V_d, during the execution. The output of the algorithm is the dynamic type valuation Q_d. We introduce Algorithm 2 in the following. Recall that in Definition 2, $Var(P)$ represents the set of all variables in P and each variable in $Var(P)$ is iterated (line 1). The dynamic type valuation of v, $Q_d(v)$, is initialized to $(\bot)$ (line 2). Each value in $V_d(v)$ is then iterated (line 3). The domain of the value q is obtained using *getDomain* function. For example, if $q = 1$ then *getDomain(q)*=`num`, as q belongs to `num` domain. Subsequently, $Q_d(v)[D]$ is updated using the value q (line 5).

4.3 Integrated Type Analysis

Algorithm 3: FixCallGraph

```
   input  : JavaScript program, P
   input  : Dynamic call graph
            edges, E_d
1  for e ∈ E_d do
2      if e ∉ C(P) then
3          add(e, C(P));
```

Algorithm 4: RectifyValue

```
   input  : Static type valuation, Q_s
   input  : Dynamic type valuation,
            Q_d
1  for v ∈ Var(P) do
2      Q_s(v)[i] ← Q_s(v)[i] ⊔ Q_d(v)[i];
```

In the integrated type analysis, we make use of dynamic type analysis in refining static type analysis to make the analysis more complete and precise. As demonstrated using the example shown in Figure 2, it is very hard for us to obtain the variable values for `code` and `res`, as well as knowing that `eval` function at line 16 invokes function f at line 4. Whereas such information can be easily obtained from dynamic analysis. Therefore, our integrated analysis makes use of the information from dynamic analysis to refine the static analysis. In particular, two important pieces of information from dynamic analysis are used to refine the static analysis.

Dynamic call graph edges, E_d. – The instrumented program will be able to capture the call graph edges that cannot be easily discovered by static analysis (e.g., the dotted edges in Figure 8).

Dynamic type valuation, Q_d. – It is the output of Algorithm 2. Dynamic analysis can discover variable values that are obtained through dynamic features of JavaScript, e.g., runtime code evaluation.

Algorithm 3 and Algorithm 4 represent two algorithms which fix the call graph information and rectify the value of static type valuation Q_s, by levaraging dynamic type valuation, Q_d. Algorithm 3 shows the algorithm to fix the call graph obtained by the static analysis, which takes a JavaScript program P and all graph edges E_d collected from the dynamic analysis as inputs. For each edge e in E_d (line 1), if e is not included in the control flow of the program P, denoted as $C(P)$ (line 2), then e is added to $C(P)$ in line 3. Then, the basic blocks of the JavaScript program P, $B(P)$, are generated again according to the modified control flow from Algorithm 3.

After fixing the call graph, the integrated type analysis algorithm is performed. The integrated type analysis algorithm is obtained by modifying Algorithm 1 with *RectifyValue*($Out[b]$, Q_d) inserted between line 8 and line 9. The function *RectifyValue* is introduced in Algorithm 4. For each variable v in $Var(P)$ (line 1), we join two types, $Q_s(v)$ and $Q_d(v)$. The reason for the joining is that, initially whenever static analysis is unsure on variable v (e.g., the value is returned by runtime generated code), then it will be assigned with ($\perp$). While for dynamic analysis, it can provide the type of variable v through the execution. After the join of two types, $Q_s(v)$ will again be used to infer types of other variables in the program. The output of the integrated type analysis algorithm will be the final type valuation.

Consider the valuation of variables `code` and `res` for the example given in Figure 2. Under pure static type analysis, the valuation is $\{code \mapsto (\bot), res \mapsto (\texttt{Err} : \{NaN, +inf, -inf\})\}$. Under pure dynamic type analysis, the valuation is $\{code \mapsto (\texttt{String} : \{''f('res')''\}), res \mapsto (\texttt{String} : \{''res''\}\}$. For integrated type analysis, the valuation is $\{code \mapsto (\texttt{String} : \{''f('res')''\}), res \mapsto (\texttt{String} : \{''res''\}, \texttt{Err} : \{NaN, +inf, -inf\})\}$. Compared to pure static or pure dynamic type analysis, integrated type analysis yields a more precise and complete type analysis.

5 Evaluation

We evaluate JSFox on popular JavaScript benchmarks and real-world Web applications to show our efficiency. In the following, we will present our evaluation in details. We have implemented JSFox in C#. The parsing and normalization of JavaScript programs are done by Esprima [17] and JS_WALA [10] respectively. We use Z3 [12] for solving the Datalog. The JavaScript program is instrumented with Jalangi [29]. In our evaluation, we seek to answer the following research questions.

RQ1. Can JSFox effectively identify type issues?
RQ2. Is it helpful in integrating static and dynamic static analysis?
RQ3. What are the root causes of these type issues?

As a baseline comparison, we compare JSFox with the state-of-art approach presented in [26]. The approach in [26] is based on pure dynamic analysis. We evaluate our tool on a popular JavaScript benchmarks collection and real-world Web applications. We introduce them in the following:

- JetStream [3]: We use JetStream version 1.1, which includes benchmarks from the SunSpider 1.0.2 [5] and Octane 2 [13]. We exclude *earley-boyer*, *typescript*, *zlib* and *coad-load* because they are obfuscated, which makes the source code difficult to analyze.
- Web Applications: Five real-world Web applications are taken from open source JavaScript frameworks and their test suites [2].

The domain cardinalities $H_i, i \in \mathbb{Z} \cap [1, 6]$ are set to 3. H_7 is set to 4, so that it can contain all exception values (i.e., *NaN*, *+Inf*, *-Inf*, *IdxErr*).

5.1 Results

Table 1 lists the JavaScript programs we use to evaluate JSFox. In the table, the first column shows the name of test cases, the second column shows the number of lines of code (LOC), which is the number of non-comment, non-blank lines of JavaScript code (excluding third-party libraries). Our evaluated programs are up to 200,000 LOC. The following two columns list the total number of detected type issues and the number of type issues that are true positive. For both columns, we report the number in the format of a/b where a and b are the number of inconsistent types and exception types respectively. When $b = 0$, we simply use a to denote $a/0$. In the column for true positive, we also include a three-tuple (s, d, i), where s and d represent the number of type issues reported by merely using Algorithm 1 (pure static type analysis) and merely using Algorithm 2

Program	LOC	Inc./Exc. type		TypeDevil
		All	True p. (s,d,i)	All
JetStream (Sunspider):				
3d-cube	167	1	1 (1,1,0)	1
3d-morph	25	0	0	0
3d-raytrace	371	0/1	0/1 (1,0,0)	0
access-binary-trees	40	0	0	0
access-fannkuch	53	0	0	0
access-nbody	172	0	0	0
access-nsieve	29	0	0	0
bitops-3bit-bits-in-byte	17	0	0	0
bitops-bits-in-byte	18	0	0	0
bitops-bitwise-and	5	0	0	0
bitops-nsieve-bits	24	0	0	0
controlflow-recursive	18	0	0	0
crypto-aes	299	0	0	0
crypto-md5	216	4	2 (0,1,1)	1
crypto-sha1	145	2	2 (0,1,1)	1
date-format-tofte	212	6	5 (2,4,1)	4
math-cordic	57	0	0	0
math-partial-sums	25	0	0	0
math-spectral-norm	41	0	0	0
regexp-dna	1702	1	1 (1,1,0)	1
string-base64	71	0	0	0
string-fasta	75	0	0	0
string-tagcloud	179	0	0	0
string-unpack-code	6	0	0	0
string-validate-input	76	0	0	0
JetStream (Octane):				
base	250	2	2 (2,0,0)	0
box2d	10,880	0	0	0
crypto	1298	1	1 (0,1,0)	1
deltablue	466	0	0	0
mandreel	271,824	0	0	0
navier-strokes	355	1	1 (0,1,0)	1
pdfjs	55,182	0	0	0
raytrace	673	0	0	0
regexp	2,070	1	1 (0,1,0)	1
richards	290	0	0	0
splay	230	0	0	0
Web Applications:				
aes	346	3	2 (0,0,2)	0
annex	857	1	1 (1,1,0)	1
HTMLParser	158	4	3 (0,0,3)	0
joomla	354	0	0	0
zurmo	193	0	0	0
Total:	349469	28	23 (8,12,8)	12

Table 1: Experiment results

(pure dynamic type analysis) respectively. *i* represents the number of type issues only reported by integrated type analysis, and missed by analysis merely using Algorithm 1 or Algorithm 2. At the last column, we list the number of inconsistent types found by [26].

As shown in the table, we have discovered 23 type issues, and out of them 8 cases can only be detected by integrated type analysis. The approach in [26] has identified 12 of them, which are all included in our pure dynamic type analysis. We explain the reason in the following. For dynamic type analysis, we use different representation from [26] to record the observed values during execution. In particular, [26] uses type graphs, and we use lattices. Nevertheless, since both approaches record the observation of variable values during an execution, this makes their reasoning power almost equivalent. In addition, there are only 5 false positives out of 349469 lines of programs, this conveys to us that the rate of false positive is low. The source of false positives has been discussed in Section 5.3.

5.2 Root Causes of Type Issues

There are 23 type issues detected by JSFox. We classify the newly detected type issues into several categories:

Redefinition argument types in function body. This happens when a function argument *a* is set to different types in a function. This redefinition normally is followed by variable declaration keyword `var`, although as explained in Section 1, `var` is not really needed. We suspect the usage of `var` could be either due to that the author of JavaScript program wants to make it explicit, or ignorance of the fact that `var` will not really provide a redefinition of the variable. We are able to detect this since variables in the same scope are merged during normalization.

Array with exception types in certain indices. This happens in *Sunspider 3d-raytrace* as introduced in Section 1. The array that contains `undefined` values is known to affect the performance [15] because certain JIT optimization cannot be applied. In addition, it can lead to maintenance problems as the developers need to be careful on handling different indices in the array.

Function arguments are treated polymorphically. *Octane base* library contains functions that are used by all *Octane* classes. We detect two functions *BenchMarkSuite* and *Benchmark* that receive different types of arguments. For *BenchMarkSuite*, it is defined as `function BenchmarkSuite(name, reference,...)`. The *reference* argument has been used for various types. For example it is used as an array type (with value `[1484000]`) in *navier-strokes* and a number type (with value `35302`) in *richards*. For *Benchmark*, it is defined as `function Benchmark(name, doWarmup,...)`. The `doWarmup` is similarly used for different types, e.g., a boolean type (with value `true`) in *navier-strokes* and function object (with function name `runRichards`) in *richards*. We are able to detect this as we perform static analysis with several *Octane* JavaScript classes concurrently. This requires the *Octane base* to handle all possible types, which adds burden to the maintenance of the library.

Function can return different types of values. *HTMLParser* returns different values for function `n` and `Obj2HTML`. In both cases, it returns -1 to signify errors and returns an object otherwise. Similarly, this adds burden for the programmer to figure out all possible values from the returned value in order to function

```
1 var a=1;
2 if (true){
3   b=2;
4 }else{
5   b=true;
6 }
7 if(b>2){
8   a=true;
9 }
```

Fig. 9: JavaScript example program

correctly. Worst still, -1 is returned to the user of *HTMLParser* when calling `Obj2HTML` with no values. A better way would be to return a `null` object.

Answer to Research Questions. We have answered **RQ1**, **RQ2** in Section 5.1. For **RQ3**, we have answered it in Section 5.2.

5.3 Analysis

In this section, we discuss the termination, soundness and completeness of our integrated type analysis, and how we group various type issues.

Termination In the following, we show that our static, dynamic and integrated analysis can always terminate. For the static type analysis (Algorithm 1), the height of lattice is bounded and H_i is assumed to be bounded. In addition, the values of $In[b]$ and $Out[b]$ for each blocks b will decrease after each round (c.f. [21] for proof). This establishes the termination of the static type analysis. For the dynamic type analysis (Algorithm 2), it always terminates since $V_d(v)$ is bounded for variable $v \in Var(P)$, with the assumption that P does not have infinite loops and terminates each time during our execution. The integrated analysis always terminates since the addition of call graph (Algorithm 3) will not affect the termination of Algorithm 1, and the addition of function *RectifyValue* (Algorithm 3) between line 8 and line 9 in Algorithm 1 will not decrease the value of $Out[b]$. Therefore, all our type analyses terminate.

False Positives The integrated type analysis is unsound due to our unsound static type analysis. Our static type analysis makes no assumption towards value returned by dynamic features (e.g., `eval`); it represents an under-approximation and no false positive will be produced in such a case. The only case where a false positive can produce is due to the path-insensitive and context-insensitive analysis; such analysis is required to deal with the path explosion problem [14]. We give an example why such analysis could result in a false positive. For example, given the JavaScript program in Figure 9, our integrated type analysis generates the type valuation Q_{int}, with $Q_{int}(a) = (\texttt{Num} : \{1\}, \texttt{Bool} : \{true\})$ and $Q_{int}(b) = (\texttt{Num} : \{2\}, \texttt{Bool} : \{true\})$. In fact, the result should be $Q_{int}(a) = (\texttt{Num} : \{1\})$ and $Q_{int}(b) = (\texttt{Num} : \{2\})$. This happens because path-insensitive analysis does not consider infeasible program paths, i.e., line 5 and line 8 will never be executed. Our evaluation in Section 5 shows that the false positive rate is low; therefore, the tradeoff of using path-insensitive and context-insensitive analysis is worthwhile.

6 Related Work

JIT Optimization. A series of works have been proposed to infer types for better Just-in-Time (JIT) compilation of JavaScript. Logozzo *et al.* [23] proposed RATA, a static analysis based on abstract interpretation, for type inference in Javascript

programs. Rastogi *et al.* [27] presented a type inference algorithm for gradually typed languages to improve the performance of programs, while preserving the run-time behaviors of unannotated programs. Hackett *et al.* [16] proposed an approach where it first statically infers types for part of the codes, and emits type-specialized code. Subsequently, the codes that are not handled statically will be accounted during runtime, and the inferred types will be updated accordingly. Aho *et al.* [7] optimized the speed of JIT compilation of Chrome V8 by removing two existing requirements that affects type predictability. Our approach can complement these approaches by supplying more type information before the JavaScript code is compiled into JIT codes.

Type Analysis for JavaScript. We first discuss related works on static type analysis for JavaScript Programs. Anderson *et al.* [9] proposed a static type inference algorithm on a subset of JavaScript, JS_0. Logozzo *et al.* [23] proposed a static type analysis on atomic types for a subset of JavaScript, *JavaScript*$^{=}$, based on abstract interpretation. Zhao *et al.* [30] proposed a polymorphic constraint-based type inference algorithm for a small subset of JavaScript, to detect the access of undefined object's member. Jensen *et al.* [20] proposed a static program analysis infrastructure for inferring type information of JavaScript programs using abstract interpretation. The dynamism of JavaScript makes the pure static type analysis hard to achieve good precision. We discuss related works on dynamic type analysis for JavaScript. Pradel *et al.* [26] proposed TypeDevil to detect inconsistent types in JavaScript programs based on the type information observed at runtime. The problem on pure dynamic analysis is due to the incompleteness of the analysis.

Other Analysis for JavaScript. Jang *et al.* [18] presented the first attempt in pointer analysis for JavaScript based on Andersen's pointer analysis. Their pointer analysis is context-insensitive and flow-insensitive, and the analysis targets on a subset of JavaScript. Madsen *et al.* [24] proposed an unsound static analysis, and performs pointer and call graph analysis declaratively using Datalog. Our Datalog analysis extends [24] with the checking of reachability of functions to reduce the search space.

7 Conclusion and Future Work

In this work, we have proposed JSFox that makes use of both integrated typing analysis – that leverages both static and dynamic typing analysis, which allows more precise and complete type analysis than any individual typing analysis can obtain. We have applied our methods in evaluating popular benchmarks and several real-world Web applications. The typing analysis has been shown to be able to identify 23 potential type issues.

As future work, we will investigate our type inference method for other analysis methods such as symbolic execution. In addition, we will also investigate other methods in combining static and dynamic analysis to provide better synergy between these analyses.

References

1. Datalog Rules for JSFox. http://tianhuat.bitbucket.org/rule.htm/.

2. Defensive javascript. http://www.defensivejs.com/.
3. Jetstream. http://browserbench.org/JetStream/.
4. Language trends on github. https://github.com/blog/2047-language-trends-on-github.
5. Sunspider. https://webkit.org/perf/sunspider/sunspider.html.
6. Tiobe index for april 2016. http://www.tiobe.com/tiobe_index.
7. W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving javascript performance by deconstructing the type system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 51, 2014.
8. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
9. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pages 428–452, 2005.
10. I. T. W. R. Center. Javascript wala, note=https://github.com/wala/JS_WALA.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
12. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.
13. G. Developers. Octane. https://developers.google.com/octane/.
14. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
15. L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2015.
16. B. Hackett and S. Guo. Fast and precise hybrid type inference for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 239–250, 2012.
17. A. Hidayat. Esprima. http://esprima.org/.
18. D. Jang and K. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1930–1937, 2009.
19. S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44, 2012.
20. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009.
21. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.
22. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
23. F. Logozzo and H. Venter. RATA: rapid atomic type analysis by abstract interpretation - application to javascript optimization. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 66–83, 2010.
24. M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Joint Meeting of the*

European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 499–509, 2013.

25. M. D. Network. Array.prototype.length - javascript — mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length.
26. M. Pradel, P. Schuh, and K. Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324, 2015.
27. A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 481–494, 2012.
28. G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 52–78, 2011.
29. K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 615–618, 2013.
30. T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *Proceedings of the 7th Symposium on Dynamic Languages, DLS 2011, October 24, 2011, Portland, OR, USA*, pages 37–50, 2011.