RC25359 (WAT1303-015) March 6, 2013
Computer Science

IBM Research Report

Effective Smart Completion for JavaScript

Max Schäfer
Nanyang Technological University

Manu Sridharan, Julian Dolby
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Frank Tip
University of Waterloo

Effective Smart Completion for JavaScript

Max Schäfer
Nanyang Technological University
schaefer@ntu.edu.sg

Manu Sridharan Julian Dolby
IBM T.J. Watson Research Center
{msridhar,dolby}@us.ibm.com

Frank Tip
University of Waterloo
ftip@uwaterloo.ca

ABSTRACT

Many modern IDEs offer a smart completion feature: when the programmer enters an expression e followed by a dot character, the IDE offers a list of properties available on e to complete the expression. For statically typed languages, this list is easily computed from the type of e and the class hierarchy. In JavaScript, the same problem is much more challenging, since expressions are not statically typed and object properties may change over time. The problem is exacerbated by the common use of complex framework libraries like jQuery and native libraries like the browser DOM.

We present a novel approach to JavaScript smart completion based on combining static and dynamic analysis. Completions are computed using a static pointer analysis enhanced with support for usage-based property inference. Frameworks are handled using a fully automatic dynamic analysis which infers API models based on the framework's test suite. We have implemented our approach in a tool called PYTHIA and evaluated it on a set of real-world JavaScript programs. On average, PYTHIA was able to suggest the desired property name 88% of the time, compared to 63% for the best open-source completion engine.

1. INTRODUCTION

In recent years, the popularity of JavaScript has increased dramatically. Having first gained notice as the enabling technology for rich in-browser applications like GMail, its use has now spread to servers [22], mobile platforms [25], and even desktop platforms like Windows 8 and GNOME.

This popularity has increased the demand for JavaScript integrated development environments (IDEs) supporting advanced features such as *smart completion*: when the user writes an expression e followed by a dot '`.`', a smart completion engine suggests appropriate property names to appear after the dot. If, e.g., e may evaluate to a string at runtime, the engine could suggest the property `length` (holding the string's length) or `substring` (to extract a substring).

Smart completion is already widely available in IDEs for Java and C#. It is particularly helpful if the programmer is not sure of the name of the property she is trying to access (a common scenario when using large, unfamiliar libraries and frameworks), or if she wants to access a property with a very long name, since typing a few characters after the dot is often enough to uniquely identify a property.

Many smart completion engines rank suggested completions, filtering out suggestions that would lead to type errors and using type-based heuristics to ensure that more likely completions appear earlier in the list of suggestions [24]. Going beyond just completing property names, some systems can even suggest more complex expressions [5, 10, 20, 24].

This paper will concentrate on the basic problem of property name completion: for a *receiver expression* e appearing somewhere in the program, we want to be able to suggest any property that e may have at runtime as a completion.

This problem is easy to solve in statically typed languages. In Java, for instance, if e has static type `C`, possible completions are immediately limited to the accessible fields and methods that `C` declares or inherits, since any other completion would cause a compile error.

In JavaScript, the same problem is much more difficult. We identify four main challenges:

(C1) JavaScript is dynamically typed, so statically determining the possible types of an expression is itself a challenging problem. Moreover, object properties can be added and removed freely at runtime, hence the set of properties of an object is not fixed.

(C2) Most JavaScript programs are built on top of complex frameworks such as jQuery [15], which tend to make extensive use of dynamic language constructs that are hard to analyze statically [29].

(C3) Typical JavaScript programs rely heavily on native libraries such as the Document Object Model (DOM) API, which is built into the browser and cannot easily be analyzed.

(C4) Smart completion should be able to deal with incomplete programs still under development. In particular, smart completion should be available in functions that are not yet called from anywhere.

Current smart completion engines for JavaScript tackle (C1) and (C4) by identifying commonly used programming idioms and then heuristically inferring the set of properties an expression may have. Additionally, they usually employ some form of *usage-based property inference*: given a receiver expression e, completions are discovered by observing what properties are accessed on e elsewhere in the program.

These heuristics tend to fail for framework code and do not apply to native libraries. Consequently, JavaScript IDEs often ship with hand-written models of commonly used framework libraries like jQuery and the DOM to deal with (C2) and (C3). While this approach works for the DOM, which is evolving only slowly, it is not practical for frameworks: there is a significant and growing number of popular frameworks, and individual frameworks are evolving very rapidly,[1] making manual model maintenance a daunting task.

We propose a novel approach to smart completion for JavaScript that combines static and dynamic analysis. A static pointer analysis is used to approximate the set of abstract objects that may flow into a receiver expression e, and to compute which properties may be available on these objects, which takes care of (C1). These properties are then suggested as possible completions on e.

Our pointer analysis ignores most of JavaScript's reflective features such as dynamic property accesses and eval. This makes the analysis simpler and faster, but also fundamentally unsound in that it may fail to model data flow arising from these features. However, due to the additional techniques described below, it works quite well in practice, and in particular much better than the heuristics employed by other IDEs.

To deal with incomplete code (C4) and missing flows due to unsoundness, we extend the basic analysis with features mimicking usage-based property inference. First, we explicitly track the *catalog* of every abstract object, i.e., the set of properties that have been either read or written on it. If the analysis has seen reads of, but no writes to, a property f on an abstract object o, f will still be entered into o's catalog and hence suggested as a completion for a receiver expression that may evaluate to o. Second, we introduce dummy objects for every local variable and parameter in the program, so that even if no flow into the variable is observed, catalogs operations can still be performed on the dummy objects. Finally, the analysis assumes that any function defined as a property of an object literal l may be invoked with l as its receiver, even if no such invocation is observed.

It remains to deal with (C2) and (C3). Pointer analysis is difficult to scale to frameworks to begin with, and for our analysis unsoundness becomes a real problem since frameworks make heavy use of reflective language features [29]. In practice, however, it often suffices for the analysis to know the properties of objects reachable from globals exposed by the framework, and return types for exposed methods.

We use a fully automatic dynamic analysis that infers an *API model* capturing this information. The key insight underlying this analysis is that most frameworks come with extensive unit test suites that exercise all important behaviors of its API methods. We run these test suites on an instrumented version of the framework, which observes property writes and function returns to create models for the values stored in object properties and returned from functions.

A similar model for the DOM can be automatically extracted from publicly available specifications. Using API models instead of the frameworks themselves and taking the DOM model into account, our pointer analysis can successfully analyze framework based JavaScript code.

[1]jQuery alone has had no fewer than seven releases (1.7.2 to 1.9.1) in the past year.

We have implemented our approach in a prototype tool named PYTHIA, which is publicly available [27]. To assess its effectiveness, we evaluated it on a suite of real-world programs, checking for every property read e.f whether f was, in fact, among the completions PYTHIA suggests. We found that, on average, this was indeed the case for 88% of all property reads, which is a significant improvement over the 63% success rate achieved by the best open-source smart completion engine. We also found that in spite of little tuning, PYTHIA could already provide interactive performance for medium-sized programs. Finally, we compared our automatically generated framework models against two hand-written models shipping with JavaScript IDEs, and found that our models provided better coverage of important APIs and contain no spurious properties.

In summary, this paper makes the following contributions:

- We describe a static pointer analysis for computing completions, including enhancements to handle incomplete programs.
- We introduce a dynamic analysis for inferring API models for JavaScript libraries, suitable for enabling smart completion in programs using those libraries.
- We describe PYTHIA, an implementation of these techniques, and present experiments showing PYTHIA to be significantly more effective at suggesting completions than state-of-the-art open-source IDEs.

The rest of this paper is organized as follows. Section 2 presents a detailed example that shows the difficulty of smart completion for JavaScript and illustrates our techniques. Then, Section 3 details the static and dynamic analyses used in our approach to smart completion. We describe our implementation in Section 4, and present an experimental evaluation of its effectiveness in Section 5. Section 6 discusses related work, and Section 7 concludes.

2. MOTIVATING EXAMPLE

We begin by showing a short example program that illustrates the main challenges in building a smart completion engine for JavaScript, and discussing how our approach addresses them.

2.1 Overview of example

Fig. 1 shows our example, a simple browser-based program that displays an HTML table and programmatically sets the background color of every td element to a dark green. It consists of three files: a miniature framework library (jQ.js), some client code for setting the background color (client.js), and an HTML page including both the library and the client code (example.html).

Framework. The framework, shown on the left in Fig. 1, is modeled after a tiny subset of jQuery. Its workings are rather intricate and will be explained in some detail because of their relevance to the smart completion problem at hand.

The main entry point of the framework is function jQ, defined on lines 2–4. It accepts a single parameter n and returns an array-like result object containing all DOM nodes (i.e., JavaScript objects wrapping HTML elements) on the current page whose tag name is n; for instance, invoking jQ('td') would return all td elements.

```
 1 (function() {
 2   function jQ(n) {
 3     return new jQ.fn.init(n);
 4   }
 5
 6   jQ.fn = {
 7     init: function init(n) {
 8       var elts = document.getElementsByTagName(n);②
 9       for(var i=0;i<elts.length;++i)
10         this[i] = elts[i];
11       this.length = elts.length;
12     },
13
14     extend: function extend(obj) {
15       for(var p in obj)
16         jQ.fn[p] = obj[p];
17     }
18   };
19
20   jQ.fn.init.prototype = jQ.fn;
21
22   jQ.fn.extend({
23     each: function(cb) {
24       for(var i=0;i<this.length;++i)
25         cb(this[i]);
26     }
27   });
28
29   window.jQ = jQ;
30 }());
```

jQ.js

```
31 <html>
32   <head><title>Example</title></head>
33   <body>
34   <table border="1">
35     <tr><td>A</td><td>B</td></tr>
36     <tr><td>C</td><td>D</td></tr>
37   </table>
38   <script src="./jQ.js"></script>
39   <script src="./client.js"></script>
40   </body>
41 </html>
```

example.html

```
42 jQ.fn.highlight = function highlight(c) {
43   this.each(function(elt) { ⑤
44     elt.style.backgroundColor = c; ⑥
45   });
46 };
47
48 function highlightNodes(n) {
49   jQ(n.name).highlight(n.color); ①, ④
50 }
51
52 function setTextColor(q){
53   if (q.textColor)
54     jQ(q.name).setTextColor(q.textColor); ③
55 }
56
57 var p = { name : 'td', color : '#449955' };
58 highlightNodes(p);
```

client.js

Figure 1: Example of a library-based JavaScript program. Desired code completions are indicated using shaded text and labeled with circled numbers.

To understand how this is done, consider the new expression on line 3. While its syntax resembles a new expression in Java, its semantics is quite different. At runtime, it creates a fresh, empty object *o*, invokes the function stored in jQ.fn.init (which is the init function defined on lines 7–12) with *o* as its receiver, and finally returns *o*. The init function, in turn, uses the DOM method document.getElementsByTagName to obtain a list of all DOM nodes with tag n. It iterates over that list (lines 9–10), copying the nodes into properties 0, 1, ... of the new object. These properties do not exist yet (remember that the new expression creates an empty object), but writing to a non-existent property in JavaScript simply creates that property. Finally, the length property is set to the number of found DOM nodes on line 11.

Observe that the init function itself is stored in property init of the object literal on lines 6–18, which in turn is stored in the property fn of jQ—functions are first-class objects in JavaScript, so they can have properties.[2] The object literal contains another function extend discussed below.

On line 20, the prototype property of the init function is set to jQ.fn. The result of this somewhat circuitous maneuver is that objects created using new jQuery.fn.init will inherit all properties of jQ.fn. Hence, adding properties to jQ.fn makes them available on all objects returned by jQ. Such properties can be installed, for instance, using the extend function (lines 14–17), which copies all properties of its argument to jQ.fn, overwriting any existing properties of the same name.

extend is used on lines 22–27 to add a function each to jQ.fn. This function takes a callback cb as its argument, and invokes it on every element stored in the receiver. Finally, on line 29, the jQ function is made available in the global scope by assigning it to the jQ property of the window object.

Client code. The right-hand side of Fig. 1 shows a small client of our jQ library, consisting of files `example.html` and `client.js`. The former is an HTML file containing a table with two rows and two columns, which loads and executes the `jQ.js` and `client.js` files by means of script tags.

Loading jQ.js will initialize our library and make the jQ function available in the global scope, as previously discussed. Loading client.js adds a function highlight to jQ.fn (lines 42–46). This function takes a color c as an argument and uses the each function to set the background color of each element in the receiver to c. On lines 48–50, a function highlightNodes is defined. This function takes an argument n, and uses jQ to create an array containing all HTML nodes of type n.name and set the color of these nodes to n.color.

Lastly, on line 57, we invoke the highlightNodes function with an object literal to set the background color of all 'td' elements in the current page to a dark green color. The setTextColor function (line 52) is not used; we include it to

[2]Function names are optional in JavaScript; we use names that match the assigned properties for expository purposes.

illustrate issues with code under development.

2.2 Smart completion in the example

We will now discuss the main challenges of implementing smart completion for JavaScript by considering what is involved in suggesting the names of the properties in the expressions labeled ①, · · · , ⑥ in Fig. 1.

Pointer analysis. JavaScript variables do not have declared types and may hold values of different types over time. To determine what properties n may have at point ①, a smart completion engine has to find out which objects it may hold at runtime. In this case, it can only hold the object literal defined on line 57, but interprocedural reasoning is needed to see this. Such reasoning requires a call graph, which is difficult to construct in JavaScript [7]. Our approach relies on a static pointer analysis to track interprocedural data flow and build a call graph at the same time.

Native libraries. The flow analysis needs to be supplemented by models of the standard library and the DOM, for which no implementation is available. For example, the expression ② has as its receiver the global variable document containing the document object. This object is not an ordinary JavaScript object but is implemented internally by the browser, so a model describing its properties is needed to be able to suggest meaningful completions.

Incomplete programs. Since smart completion is usually performed on code that is still under development, it has to deal with incomplete code. For instance, expression ③ occurs in method setTextColor, which is not yet invoked anywhere. A standard pointer analysis cannot infer any properties for its parameter q, since so far no objects flow into q. However, the conditional test on line 53 indicates that q may have a property textColor, so it would be reasonable to suggest this property as a completion on the next line.

To support this kind of *usage-based property inference*, our analysis maintains, for every abstract object, a catalog of properties that are read or written on that object anywhere in the program. Additionally, it introduces dummy objects for every parameter and local variable in the program to make sure that these variables contain at least one abstract value. In particular, q is assigned such a dummy object o_q. The property read on line 53 adds textColor to the catalog of o_q, and hence textColor is suggested as a completion at ③.

Handling frameworks. For expression ⑤, the receiver expression this must hold an object returned by jQ. As discussed above, such objects inherit all properties of jQ.fn, in particular extend and each. But while extend is set up directly in the definition of jQ.fn on lines 6–18, each is added dynamically by the call to extend on lines 22–27. In general, sound and precise reasoning about reflectively added properties is still beyond state-of-the-art pointer analyses [29].

In practice, such reflection occurs mostly in framework code, not client code. If we can create a model of the framework that "unrolls" reflective code and makes it explicit that jQuery.fn has a property each, then we can avoid having to reason about intricate reflective functions like extend.

We use a dynamic analysis, described in more detail in Section 3.1, that builds an API model based on the framework's test suite. Basically, the analysis infers a model for every function and every object literal in the framework, as well as a common model for all instances of a given function f (i.e., all objects created by invoking new f()). For any object o, the model of o lists all properties p of o that were written during the test run, together with information about the models of all values that were written into p. For a function f, it additionally records information about the models of all values that were returned from f. The model can itself be encoded as a JavaScript program to be analyzed together with client code.

```
59 var init_model = function () {
60   this.length = 1;
61 };
62
63 var fn_model = {
64   init: init_model,
65   extend: function(){},
66   each: function(){}
67 };
68
69 init_model.prototype = fn_model;
70
71 var jQ_model = function () {
72   return new init_model();
73 };
74 jQ_model.fn = fn_model;
75 window.jQ = jQ_model;
```

jQ-model.js

Figure 2: Automatically inferred API model for jQ.js.

For our example, let us consider a single test for jQ.js that consists of the function call jQ('body'). Fig. 2 shows the model inferred for jQ.js when executing this test, slightly edited for readability. Here, init_model is the model of function init, fn_model is the model of the object literal stored in jQ.fn, and jQ_model is the model of function jQ. During the test run, jQ was observed to return an instance of init, which is reflected in its model. Function init did not return anything, hence its model does not have a return statement, and similar for extend and each. When the call to extend on line 22 is executed, the dynamic analysis observes that a function is written into property each of jQ.fn; consequently, its model fn_model is updated to contain a property each. Finally, note that while the assignment to this.length on line 11 is reflected in init_model, the assignment to this[0] on line 10 is not, since it is irrelevant for smart completion.

This API model enables inference of the desired completions. For ④, we reason that: (i) function jQ returns a new init_model object (line 72), whose prototype is fn_model (line 69), and (ii) jQ.fn is aliased with fn_model (line 74). Therefore, the assignment to jQ.fn.highlight on line 42 assigns to fn_model.highlight, so highlight is inherited by all result objects of jQ, allowing us to suggest highlight as a completion for ④. Similarly, we can see that the receiver of function highlight must have fn_model as its prototype, which enables us to complete this.each at ⑤.

In some cases, however, the API model is too imprecise: expression ⑥ occurs in a method that is invoked by a callback from the framework on line 25. The model does not capture this callback, and hence we are unable to suggest a suitable completion. In practice, however, usage-based property inference often enables us to suggest some completions in such cases anyway.

3. ANALYSES

In this section, we present the two key analyses that comprise our smart completion engine. First, we present our dynamic analysis for generating API models, which automatically infers models for objects and functions based on dynamic value flow. Then, we describe our static pointer analysis for computing possible completions, which leverages the API models to better handle complex libraries and includes special support for usage-based property name inference.

3.1 Generating framework models

Overview. Given a framework library like jQuery, we want to automatically compute an *API model* for the library, enabling better smart completion for its clients. A JavaScript library typically exposes its API via global variables created during script loading. Hence, an API model for a library should describe the objects reachable from the library's globals. To enable smart completion, it should record, for every such object, the names of all its properties, and recursively include models for their values. For functions f reachable from the globals, the API model should additionally provide models for the function's return value to enable smart completion on results of calls to f.

Under JavaScript's dynamic typing discipline, the same property may hold values of different types over time, and functions may return values of different types depending on how they are invoked. Hence, our models for property values and function return values will, in general, be *sets* of models (similar to union types), one for every possible value stored in the property or returned from the function.

To automatically compute such models, we run an instrumented version of the framework, where each runtime object is tagged with its model. These models are updated as property writes or function returns are observed. After the execution has finished, the models for all objects reachable from the global scope are assembled into JavaScript code that can be analyzed instead of the framework code itself.

Since the model is entirely based on runtime information, it is in general incomplete, and its quality depends crucially on the coverage achieved during the training run. However, most popular frameworks come with extensive unit test suites that aim to exercise all of their important behaviors. Running these test suites on the instrumented framework is thus likely to yield high quality models;[3] in fact, our evaluation showed our models to be more complete than hand-written counterparts (see Section 5).

We will now describe in more detail how the dynamic analysis builds models, and how these models are serialized to JavaScript.

Building models. The dynamic analysis constructs models for object literals and functions: for an object literal appearing at location l in the program text, there is a model $\texttt{Obj}(l)$, and for every function f there is a model $\texttt{Fn}(f)$. Additionally, there is a model $\texttt{New}(f)$ representing all objects created by using the new operator on function f. When executing the instrumented program, objects are tagged with their model upon creation as shown in Fig. 3, where we use the notation $[\![e]\!]$ to mean e's runtime value. Note that the same object literal or function expression may be evaluated multiple times, yielding different runtime objects, but all these objects share a single model.

$$[\![\texttt{\{\}}^l]\!].\mathit{model} = \texttt{Obj}(l)$$

$$[\![\texttt{function f() \{...\}}]\!].\mathit{model} = \texttt{Fn}(f)$$

$$\frac{[\![\texttt{y}]\!].\mathit{model} = \texttt{Fn}(f)}{[\![\texttt{new y(...)}]\!].\mathit{model} = \texttt{New}(f)}$$

Figure 3: Tagging objects with their models

Statement	Constraint
`x[y] = z`	$\dfrac{[\![\texttt{x}]\!].\mathit{model} = M_x \quad [\![\texttt{y}]\!] = \texttt{"s"} \quad [\![\texttt{z}]\!].\mathit{model} = M_z}{M_z \in M_x.\mathit{props}[\texttt{"s"}]}$
`return x` in function `f`	$\dfrac{[\![\texttt{x}]\!].\mathit{model} = M_x}{M_x \in \texttt{Fn}(f).\mathit{ret}}$

Figure 4: Updating models during execution.

For every model M, the analysis maintains a mapping $M.\mathit{props}$ from property names to sets of models: for a property name p, $M.\mathit{props}[p]$ should contain the models of all values that were observed to flow into property p of any object with model M. Additionally, every function model F maintains a set $F.\mathit{ret}$ describing the observed return values of all functions modeled by F.

Fig. 4 shows how these mappings are updated at runtime: when a property write x[y] = z is encountered, we need to ensure that the model M_z of z is contained in the set $M_x.\mathit{props}[\texttt{"s"}]$, where M_x is the model of the object stored in x and y evaluates to "s".[4] Similarly, when we observe execution of a statement return x in a function f, we need to ensure that the model M_x of x's value is contained in $\texttt{Fn}(f).\mathit{ret}$.

Serializing models. Our final API model consists of the models of all objects reachable from the global scope. For integration with the pointer analysis, we serialize the API model to JavaScript source code like that in Fig. 2.

Each model is represented as a JavaScript object assigned to a global variable for easy reference. An object literal model $\texttt{Obj}(l)$ is itself represented as an object literal (cf. fn_model in Fig. 2), with property names and models encoded as properties of the literal. Similarly, a function model is also represented as a function, with a return statement to encode its return type (cf. function jQ). Any properties the function may have are encoded as top-level property writes (cf. line 74). A model $\texttt{New}(f)$ is represented as an expression

[3]Observe that we only require test suites for framework code; client code is analyzed entirely statically.

[4]A write x.f = z can be seen as syntactic sugar for x['f'] = z.

new f(), where f is the global variable containing the model for f. Properties of function instances are encoded as property writes on this inside the constructor function (cf. line 60). Finally, sets of models are represented as disjunctions using the || operator: since our pointer analysis is flow insensitive, this amounts to a non-deterministic choice.

We perform two optimizations to make the resulting JavaScript code more compact and readable. First, we inline models that are only referenced once, as is done for the models of extend and each on lines 65 and 66 in Fig. 2. Second, we merge models that are structurally identical, which significantly reduces code size in practice.

3.2 Static Pointer Analysis

Given the API models described in Section 3.1 and the client code, we employ an enhanced static pointer analysis to determine what completions to suggest.

Our basic analysis is a field-sensitive, context- and flow-insensitive Andersen-style points-to analysis [2] for JavaScript, similar to analyses described in the literature [8]. Our heap abstraction is coarser than that typically employed in an Andersen-style analysis, as only one abstract object is used to represent all instances of a function, matching the models inferred by our dynamic analysis. The analysis still uses separate abstract objects per object literal and function, however.

Similar to some previous work [7, 8], we eschew a sound handling of dynamic property accesses. Instead, we treat dynamic property reads and writes as accessing a special "unknown" property similar to how array elements are usually modeled in pointer analysis. This is unsound as we do not model the potential overlap between the unknown property and other named properties. However, in practice many non-trivial uses of dynamic property accesses are found in framework code. Since we analyze API models of frameworks instead of their implementations, sound handling of such accesses is much less critical.

Like other such analyses [32], our pointer analysis maintains a *catalog* of properties observed to exist for each abstract object. This makes computing possible completions for a receiver expression e straightforward: we simply take the union of the catalogs of all abstract objects in e's points-to set.

Enhancements. Even with good API models for frameworks, analysis unsoundness and other factors remain that cause the basic analysis to miss completions. One important factor is "dead" code, i.e., functions which, according to the analysis, are never called. Such functions may be genuinely unreachable, e.g., if the application is still incomplete (see ③ in Fig. 1). They may also just appear to be unreachable, either due to unsoundness in the analysis or due to incompleteness of the API model (see ⑤ in Fig. 1) For such code, a pointer analysis-based approach to completion is ineffective, as the analysis cannot observe which objects may flow into function parameters and from there into other variables.

To improve coverage of completions missed by the standard pointer analysis, we add three special rules to the analysis to perform a limited form of usage-based inference of properties. First, an abstract object's catalog is not only updated on property writes, but also on property reads. So, if the analysis observes a read of property f on abstract object o, it adds f to o's catalog, and will suggest f as a completion for expressions that may evaluate to o. Second, we add a fresh dummy abstract object to the points-to sets of all formal parameters and local variables, thus ensuring that they point to at least one abstract object. With these two changes, the analysis can learn from property accesses in dead code and provide improved completions.

Third, we take inspiration from other smart completion algorithms [23] to enhance the handling of functions appearing in object literals like init and extend in Fig. 1. We observe that such functions are often invoked with that object literal as their this parameter, although the analysis may not observe such an invocation due to issues outlined above. Hence, the analysis automatically adds the abstract object for a literal l to the points-to set of the this parameter of any function assigned to a property of l. With this enhancement, all of l's properties will be suggested as completions for this in the attached functions, and furthermore the pointer analysis will propagate suggestions based on property accesses on this among the attached functions. For example, consider the following object literal:

```
{ m: function() { this. n (); }, n: function() { } }
```

With this enhancement, n will be suggested as a completion on this within m, even if m is not yet invoked in the program.

4. IMPLEMENTATION

We have implemented our approach in a prototype tool called PYTHIA, which is available online [27]. PYTHIA is itself written in JavaScript and makes use of JavaScript analysis utilities from WALA [32].

To generate API models, PYTHIA provides a source-level instrumentation tool which rewrites a given JavaScript program to compute the models described in Section 3.1. Once the framework has been instrumented, it can be substituted for the original framework in its test suite, which is then run to obtain the model. The test suites for most frameworks require some non-trivial setup and have to be run in a browser, so this step has to be performed manually.

Modeling the DOM is a notoriously difficult problem for static analyses [13]. For the purposes of property completion, however, it is sufficient to specify which properties are available on different DOM objects, and what types of objects may be returned by DOM methods. This is, of course, the same kind of information provided by the auto-generated framework API models, and it can readily be derived from the official WebIDL specification of the DOM [34]. We manually added specifications for a number of commonly-used non-standard properties provided by Internet Explorer.

PYTHIA is not yet integrated into an IDE; instead, it can be run under a JavaScript shell such as node.js. As its input, it expects a completion problem, expressed as an ordinary JavaScript program with exactly one property access of the form e.$$f. After parsing the program and running the pointer analysis described in Section 3.2, PYTHIA checks whether a property named f appears in the catalog of any abstract object that expression e may evaluate to, i.e., whether the desired property is among the suggested property names.

Currently, frameworks must be manually replaced by their models before PYTHIA is run, but in the future we plan to keep a database of pre-generated models for common frameworks. The pointer analysis can then check whether a model is available for a given input file (for instance by computing a hash), and if so, analyze the model instead.

5. EVALUATION

In this section, we describe an experimental evaluation of our smart completion approach. We evaluated our approach in terms of practical usefulness, compared it against a state-of-the-art open source implementation of property completion for JavaScript, and assessed the impact of the individual techniques comprising our approach.

To find real-world completion problems, we collected nine open-source JavaScript programs (further discussed in Section 5.1). For every property read expression of the form `e.f` in these programs, we considered the problem of completing the property name `f` given just the receiver expression `e`.[5] To more closely approximate a development setting, we furthermore deleted all code following `e.f` in its enclosing function, as other evaluations of smart completion systems have done [24].

Given this set of completion problems, we evaluated the usefulness of our approach with respect to three criteria:

(EC1) Completeness: How many completion problems can our approach solve?

(EC2) Scalability: How long does it take to compute the list of suggestions?

(EC3) Precision: How many property names are suggested altogether?

Since we do not yet consider the problem of ranking suggested completions, we did not measure the position at which the desired property name occurs among the suggestions. Clearly, a completion algorithm could score very highly on (EC1) by simply suggesting all properties occurring anywhere in the program. The purpose of (EC3) is to rule out this trivial solution.

We also evaluated the impact of the individual techniques comprising our approach:

(EC4) How do individual analysis features (framework modeling, DOM model, usage-based inference) contribute to the overall completeness?

Finally, we assessed the quality of our framework models:

(EC5) How complete are our auto-generated framework models compared to hand-written models used by other IDEs?

For (EC1)–(EC3), we compared Pythia against the property completion algorithm shipping with Eclipse Orion 1.0 [23] (also used by VMWare's Scripted editor [28]), which we ran on the same completion problems. We also compared against three other popular JavaScript IDEs, namely Aptana Studio 3 [3],[6] Eclipse JSDT 1.4.1 [16] and Komodo Edit 8 [18], on a subset of problems. All of these IDEs use AST-based heuristics to implement property completion, and as far as we can tell, none of them implements a full pointer analysis. Another JavaScript IDE, Adobe Brackets [4], also provides auto-completion via a plugin. However, we found (via testing and code inspection) that their technique always suggests a ranked list of all properties in the program under consideration, making it hard to compare against our approach.

On average, Pythia was able to complete 88% of all property reads in our subject programs, significantly outperforming Orion (63%) and the other IDEs. While Pythia was in general slower than a purely AST-based heuristic, it was already fast enough to be used in an interactive setting: on a modern machine, completions were computed in less than two seconds for all but the largest programs.[7]

Program	Underlying Framework	LOC	# Compl. Problems
3dmodel	none	4880	305
beslimed	MooTools 1.2b2	3995+755	740
coolclock	jQuery 1.7.2	6483+416	239
fullcalendar	jQuery 1.7.1	6383+5883	1222
htmledit	jQuery 1.3.2	2953+653	473
markitup	jQuery 1.6.2	5939+532	318
pacman	none	3513	1816
pdfjs	none	31694	200
pong	jQuery 1.3.2	2953+693	550

Table 1: Subject programs

5.1 Subject programs

Table 1 lists our subjects, which are nine medium to large browser-based JavaScript applications covering a number of different domains, including games (*beslimed*, *pacman*, *pong*), visualizations (*3dmodel*, *coolclock*), editors (*htmledit*, *markitup*), a calendar app (*fullcalendar*), and a PDF viewer (*pdfjs*).[8]

For each subject program, we list the framework that they rely on, if any. Out of nine subject programs, six use a framework, of which five use jQuery and one uses MooTools. This distribution is roughly representative of real-world framework usage: according to a recent survey [31], 38.5% of all websites use no framework at all, 55.8% use jQuery, and 4.8% use MooTools. Crucially, both jQuery and MooTools provide extensive unit test suites that we used to automatically generate framework models.

The table also shows the size of the subject programs measured in non-blank, non-comment lines of code as determined by the `cloc` utility; for framework-based programs, we list their size as the sum of the size of the framework and the size of the client code. Note that in all cases the framework is larger than the client code, in most cases significantly so.

Finally, the last column lists the number of completion problems we generated from every program. For *pdfjs*, the large size of the benchmark would have led to an excessive number of completion problems, so we selected 200 problems at random.

[5]We do not consider property writes since they often refer to non-existent properties; recall that in JavaScript, a write to a non-existent property creates it on the fly.

[6]We used the build with ID 3.3.2.201302081546.

[7]All experiments were performed on a Toshiba Portégé R930 laptop with an Intel Core i7-3520M CPU, using version 3.11.10 of the V8 JavaScript engine with the standard maximum heap size of 1.4GB.

[8]These are the same benchmarks used in Feldthaus et al. [7], minus the *flotr* benchmark. We had to exclude *flotr* since the unit tests for the underlying Prototype framework are split among multiple HTML files, and our implementation cannot yet merge API models from multiple executions.

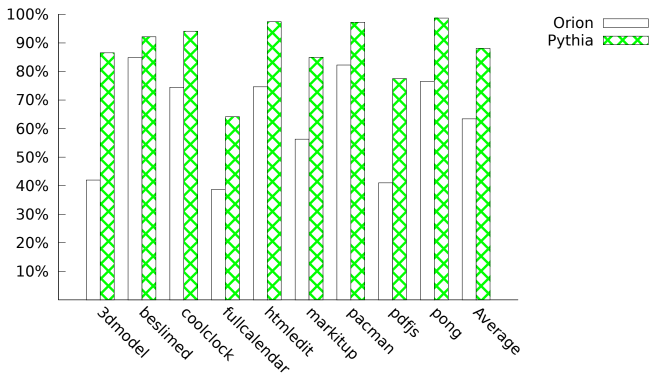

Figure 5: Percentage of successful completions across all subject programs (Pythia vs. Orion).

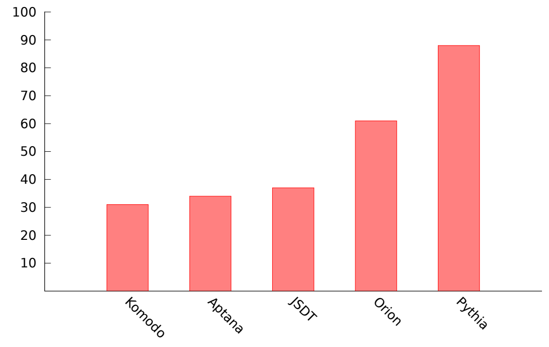

Figure 6: Number of successful completions on 100 random problems (Pythia vs. several IDEs).

5.2 Completeness (EC1)

To evaluate the usefulness of our approach, we measured the percentage of completion problems for which it could suggest the desired property, and compared this against Orion's completion algorithm. The results are shown in Fig. 5: on average, PYTHIA was able to solve 88% of the completion problems; for all but one subject program, it solved more than 75%, and for five of them it solved more than 90%. Our relatively poor showing on *fullcalendar* is due to a few crucial uses of dynamic property reads to access central data structures in that program, which our analysis unsoundly ignores.

In all cases, PYTHIA did significantly better than Orion, which solved 63% of completion problems on average. The program where we offer the least improvement is *beslimed*: PYTHIA only adds seven percentage points to Orion's result. On *3dmodel*, on the other hand, we offer 87% successful completions compared to Orion's 42%. We further discuss the factors contributing to this improvement in Section 5.5.

The open architecture of Orion made it easy to run its completion algorithm in batch mode, enabling a comparison on a large number of problems. For other IDEs, it was not as easy to run the completion algorithm on its own. Instead, we manually tested the IDEs on 100 randomly selected completion problems from our benchmarks. As shown in Fig. 6, Komodo Edit was able to suggest the desired property in 31 cases, Aptana in 34, and JSDT in 37 out of 100. For comparison, Orion could handle 61 cases, and PYTHIA was successful in 88 cases, consistent with our average improvements over Orion. These results also suggest that Orion's completion algorithm is the state of the art among open-source JavaScript IDEs.

5.3 Scalability (EC2)

Pointer analyses for JavaScript tend to suffer from scalability problems, particularly for framework-based applications [29]. PYTHIA, however, scaled fairly well: it took less than 0.5 seconds to extract constraints on all benchmarks except *pdfjs*, and less than 1 second to solve them. On *pdfjs*, which is much bigger than the other benchmarks, constraint extraction took up to 1.5 seconds, and solving the constraints up to 21 seconds. Note that scalability was significantly improved by analyzing API models instead of actual framework code, to be detailed in Section 5.5.

Our results show that PYTHIA is already scalable enough to be used on small to medium framework-based programs, though more work is needed to scale to large programs. We currently build constraints from scratch every time, but further engineering would allow them to be built incrementally. In an IDE setting, this would mean constantly updating constraints in the background; then, when a completion is requested, they would only need to be solved.

5.4 Precision (EC3)

Currently, our implementation does not filter the set of suggested property names by context and it does not rank the suggestions, as these are essentially independent problems that have been studied before in the context of typed languages. Nevertheless, it is important to assess whether our flow analysis is precise enough to avoid suggesting an unreasonably large number of properties.

Across all completion problems in our subject programs, the median number of properties suggested by PYTHIA was 86. (For comparison, our models of the JavaScript standard library and the DOM, which are included in every analysis, already contain 853 unique property names.) For 220 out of a total of 5863 completion problems, the number of suggestions exceeded 500, with a maximum of 730 suggestions in one case. We manually examined these cases, which turned out to fall into three categories: (i) completion on the result of a highly polymorphic DOM or framework function (213 cases), (ii) completion on the result of a dynamic property read (six cases), (iii) completion on `window` (one case).

Category (i) is handled poorly by our analysis since it is context-insensitive. Highly overloaded functions like `$` in jQuery may return values of different types depending on their invocation context, but our API model merges all of them into a single return type, which as a result becomes quite coarse. Adding a moderate amount of context sensitivity to model inference may alleviate this problem. Category (ii) is a result of our treatment of dynamic property reads and writes, which are all interpreted as accessing one and the same "unknown" property; in some cases, this leads to significant imprecision. Finally, the large number of suggestions for Category (iii) is simply a consequence of JavaScript semantics: `window` is both an HTML element and the global object, and hence has a large number of properties that we all suggest as completions.

Note that in all but ten of these 220 cases, the desired property was among the suggestions. The cases where it was not all arose on *fullcalendar*: in seven cases, the property had been written using a dynamic property write, which

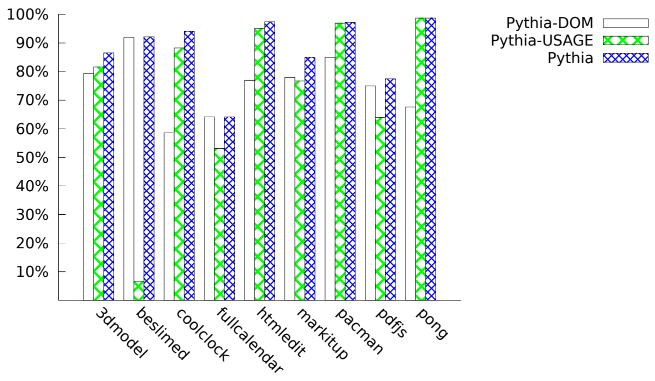

Figure 7: Percentage of successful completions with different configurations.

our pointer analysis does not fully track, and in three cases the property seemed genuinely absent.

In summary, in the small number of cases where a large number of properties were suggested, the cause was usually analysis imprecision.

5.5 Impact of individual features (EC4)

Recall that PYTHIA uses three major techniques for improving the completeness of the basic pointer analysis: (i) a DOM model, (ii) automated summarization of framework libraries, and (iii) additional rules to implement usage-based property inference. In this subsection, we show that all three techniques were needed to achieve satisfactory completeness, and that they, in fact, complemented each other.

The most immediate impact is due to (ii): if framework libraries were not summarized and PYTHIA was run directly on the unmodified subject programs, the pointer analysis failed to complete within 30 seconds on all framework-based programs, which would make it unusable in practice.

To assess the impact of (i) and (iii), we ran PYTHIA in two special configurations, PYTHIA$^{-\mathrm{DOM}}$ and PYTHIA$^{-\mathrm{USAGE}}$. In the former, the pointer analysis omitted the DOM model, while in the latter, it omitted the extra rules for usage-based inference.

Fig. 7 compares the completeness achieved by these two configurations and by the standard configuration on our subject programs. Leaving out the DOM model made a big difference on *coolclock* (36 percentage points) and *pong* (31 percentage points): although these two programs use jQuery, they contain crucial bits of code that access the DOM directly, making a DOM model indispensable. In fact, for these two benchmarks PYTHIA$^{-\mathrm{DOM}}$ did worse than Orion, which has only a rudimentary DOM model. On other benchmarks such as *fullcalendar* and *beslimed* the difference was negligible, since they perform most of their DOM accesses through the framework. On average, the DOM model improved completion success rate by 13 percentage points.

Turning off usage-based inference rules made a very big difference for *beslimed*, our only MooTools-based benchmark. Recall that our automatically generated models contain no information about callbacks from framework functions into user code. MooTools relies on such callbacks, and hence replacing it with the auto-generated model lost a lot of data flow information. Fortunately, however, our usage-based inference rules were able to recover most of this missing flow.

For the jQuery-based programs, the difference was much less marked, since they do not rely on callbacks as heavily.

In summary, framework modeling seems essential for a pointer analysis-based approach. The other features provided important improvements across the board, though their efficacy depends on framework usage and programming style.

5.6 Quality of framework models (EC5)

To further study the quality of our auto-generated framework models, we compared our models against the hand-written jQuery models shipping with Komodo Edit 8 (which models jQuery 1.6.1) and Aptana Studio 3 (which models jQuery 1.6.2). The size of these models precluded a detailed manual comparison; instead, we compared only the external API, i.e., the set of properties available on the `jQuery` function itself and on `jQuery` result objects (i.e., objects returned by invocations of `jQuery`).

The model provided by Komodo Edit contains 216 API properties. It does not distinguish between the `jQuery` function and result objects, and suggests the same set of properties on either, which is quite imprecise. Of these 216 properties, 41 are spurious, i.e., they do not, as far as we can see, actually occur either on the `jQuery` function or on `jQuery` result objects. Our auto-generated model for jQuery 1.6.1 contained a further 61 API properties not present in Komodo's model. We verified that these properties did, in fact, occur on either the `jQuery` function or on `jQuery` result objects, although in five cases their name started with an underscore, suggesting a private non-API method.

The model provided by Aptana Studio lists only 176 API properties. It does distinguish between the `jQuery` function and result objects, and none of the listed properties are spurious. However, there are 60 additional valid properties that our model covers which are not listed in Aptana's model.

For the properties that do occur in the models, both Komodo and Aptana provide detailed type information and documentation, which our auto-generated models do not provide. However, our models are clearly more complete than these hand-written models. Maintaining a hand-written model is a considerable effort, as evidenced by the fact that both models lag behind the most recent jQuery version (1.9.1 at the time of writing), whereas our models can be generated by simply running the framework's test suite.

Since our framework models are valid JavaScript code, it is possible to use them with another smart completion engine. Whether this improves completeness, however, depends on the heuristics used by the engine in question. For Orion, for instance, our models seemed to work against the completion algorithm: when they were analyzed together with client code, no completions were found at all.

5.7 Summary and threats to validity

In summary, we have shown that PYTHIA provides significant improvements over the state of the art. On average, it solved 39% more completion problems across a suite of nine real-world programs than Orion, which provides the most advanced completion support among current open source JavaScript IDEs. While PYTHIA is not highly tuned, performance was respectable and fast enough for interactive use on small to medium size programs. No filtering or ranking of properties is done at the moment, but we have shown that the number of properties suggested is generally reasonable, with a few outliers due to analysis imprecision. All major features of our approach have been shown to be useful in practice, with varying efficacy on different programs. Finally, we have shown that our auto-generated models for jQuery are more complete than the hand-written models shipping with two other JavaScript IDEs.

One threat to the validity of this evaluation is clearly in the selection of programs. We only consider browser-based applications, so it is possible that our results do not carry over to other kinds of JavaScript programs. Most of our subject programs use jQuery, with only one program using another framework. However, this is realistic in that recent data [31] suggests that less than 20% of websites use a framework other than jQuery, so our approach should be applicable to most real-world, browser-based JavaScript code.

A second threat comes from our comparison to other IDEs. We only performed one detailed comparison (against Orion), and more superficial comparisons against three other IDEs. In particular, we restricted our attention to open source IDEs. Proprietary IDEs like IntelliJ IDEA [12] and Microsoft Visual Studio 2012 [21] also provide JavaScript editing support with smart completion. It is possible that their smart completion is more advanced than what is available in the IDEs we surveyed, but since their implementations are closed source and their algorithms (to the best of our knowledge) unpublished, it would be hard to discover what approach is being used.

6. RELATED WORK

The most closely related work is that of Madsen et al. [19], who present an improved pointer analysis for analyzing JavaScript applications that rely on complex frameworks and native libraries. As one possible application, they show how to use their analysis to perform smart completion. Their technique for summarizing frameworks is entirely usage-based, a particularly useful approach for cases where no JavaScript source is available for the framework. In contrast, we generate API models based on instrumentation of framework code and dynamic analysis where source code is available, and rely on specifications for native libraries. Our usage-based inference (see Section 3.2) is simpler and less general than theirs, as we only introduce dummy objects ("symbolic locations" in their terminology) for parameters and local variables; their system can introduce dummy objects in more cases, e.g., for object properties and prototypes. Since smart completion is only one application of their system, they did not perform as extensive an evaluation as we did.

Perelman et al. [24] and Tihomir et al. [10] present more ambitious code completion techniques that consider more context than the receiver expression and try to complete other code fragments, like arguments for function calls. These approaches depend heavily on type information for ranking alternatives, which makes a direct application to JavaScript problematic. However, our pointer analysis essentially infers types of variables and properties, so it would be interesting to explore whether our analysis could be used as a basis for porting their techniques to JavaScript.

Sound and precise static analysis of real-world JavaScript programs is still an unsolved challenge [8, 9, 14, 29, 30]. Consequently, there has been some interest in unsound but more scalable techniques. Recently, Feldthaus et al. [7] presented an unsound static analysis to compute approximate call graphs for use in IDE services. Their pointer analysis only tracks function objects, which, while sufficient for building call graphs, is not suitable for smart completion. Their pointer analysis is simple enough to analyze framework implementations directly, while our more thorough analysis requires framework API models to scale.

Another strand of work has investigated combined static-dynamic analysis. Kneuss et al. [17] present a flow-sensitive static type inference for PHP that can be initialized with a particular dynamic state, aiding precision. Their dynamic technique only collects information for a single runtime state, which would be insufficient for constructing API models. Their static analysis uses a more precise heap model than ours, with one abstract object per runtime object; we have not observed a need for such precision for smart completion, and it could hurt the scalability of our pointer analysis.

An et al.'s work on type inference for Ruby [1] is broadly similar to ours: they use dynamic analysis to infer subtyping constraints between the types of constants and program variables (parameters, local variables, and object fields). Types inferred from these constraints can be shown to be sound given method-level path coverage. Our API model inference cannot provide such soundness guarantees, since it attaches metadata directly to runtime objects without tracking which variables they flow into. In practice, this seems good enough for smart completion and allows us to achieve interactive performance, which their system seems unable to deliver.

Hackett and Guo's work on type inference for JavaScript just-in-time compilation [11] employs a different blend of dynamic and static techniques. While we use dynamic analysis to aid static analysis, they do a static analysis with unsound assumptions first, and then check dynamically if the assumptions are violated. They aim to enable type-based optimizations (e.g., when both operands of + are integers), whereas we target smart completion with usage-based inference.

Wei and Ryder [33] propose a combined static-dynamic taint analysis of JavaScript programs, which leverages dynamic information to (unsoundly) handle tricky JavaScript constructs like `eval`. Their dynamic analysis does not collect information on the runtime types of objects and functions, and hence would be insufficient for constructing API models.

Recent work on static type systems for JavaScript [6, 26] differs from ours in aiming to be sound for complex JavaScript idioms. This requires much heavier-weight machinery such as SMT solvers [6]. Our focus is on effective, lightweight techniques for smart completion on real-world JavaScript programs, without the requirement of soundness.

7. CONCLUSIONS

We have presented an effective approach to smart completion for JavaScript. A static pointer analysis enhanced with usage-based property inference tracks data flow to determine possible completions. Complex frameworks are handled by inferring API models using dynamic analysis over the framework test suites. We have implemented our approach in a tool named Pythia, and have presented an experimental evaluation showing significant improvements over the completion algorithms of state-of-the-art open-source IDEs.

In future work, we plan to enrich our API models with more information about callbacks and more precisely model highly polymorphic functions. We will also investigate using our API models to perform other kinds of code completion for JavaScript [10, 20, 24] that thus far rely on static types.

References

[1] An, J. D., Chaudhuri, A., Foster, J. S., and Hicks, M. Dynamic Inference of Static Types for Ruby. In *POPL* (2011).

[2] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[3] Aptana Studio. `http://aptana.com`.

[4] Adobe Brackets. `http://github.com/adobe/brackets`.

[5] Bruch, M., Monperrus, M., and Mezini, M. Learning from examples to improve code completion systems. In *FSE* (2009).

[6] Chugh, R., Herman, D., and Jhala, R. Dependent types for JavaScript. In *OOPSLA* (2012).

[7] Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *ICSE* (2013).

[8] Guarnieri, S., and Livshits, V. B. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium* (2009).

[9] Guarnieri, S., and Livshits, V. B. Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications. In *WebApps* (2010).

[10] Gvero, T., Kuncak, V., Kuraj, I., and Piskac, R. On Complete Completion using Types and Weights. Tech. Rep. 182807, EPFL, 2012.

[11] Hackett, B., and Guo, S. Fast and precise hybrid type inference for JavaScript. In *PLDI* (2012).

[12] IntelliJ IDEA. `http://jetbrains.com/idea/`.

[13] Jensen, S. H., Madsen, M., and Moeller, A. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *FSE* (2011).

[14] Jensen, S. H., Moeller, A., and Thiemann, P. Type Analysis for JavaScript. In *SAS* (2009).

[15] jQuery Framework. `http://jquery.com`.

[16] Eclipse JSDT. `http://eclipse.org/webtools/jsdt`.

[17] Kneuss, E., Suter, P., and Kuncak, V. Runtime Instrumentation for Precise Flow-Sensitive Type Analysis. In *RV* (2010).

[18] Komodo. `http://activestate.org/komodo-edit`.

[19] Madsen, M., Livshits, B., and Fanning, M. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. MSR TR 2012-66, Microsoft Research, 2012.

[20] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI* (2005).

[21] Microsoft Visual Studio 2012. `http://www.microsoft.com/visualstudio/`.

[22] Node.js. `http://nodejs.org`.

[23] Eclipse Orion. `http://orionhub.org`.

[24] Perelman, D., Gulwani, S., Ball, T., and Grossman, D. Type-directed completion of partial expressions. In *PLDI* (2012).

[25] PhoneGap. `http://phonegap.com`.

[26] Politz, J. G., Guha, A., and Krishnamurthi, S. Semantics and types for objects with first-class member names. In *FOOL* (2012).

[27] Pythia. `http://github.com/ecspat/pythia`.

[28] Scripted. `http://github.com/scripted-editor/scripted`.

[29] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP* (2012).

[30] Vardoulakis, D., and Shivers, O. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP* (2010).

[31] W^3 Techs. Usage of JavaScript Libraries for Websites. `http://w3techs.com/technologies/overview/javascript_library/all`, March 2013.

[32] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

[33] Wei, S., and Ryder, B. G. Practical Blended Taint Analysis for JavaScript. Technical report, Virginia Tech, 2013.

[34] Document Object Model Level 3. `http://www.w3.org/DOM/DOMTR#dom3`.