Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications

Simon Holm Jensen[*]
Aarhus University
simonhj@cs.au.dk

Magnus Madsen[*]
Aarhus University
magnusm@cs.au.dk

Anders Møller[*]
Aarhus University
amoeller@cs.au.dk

ABSTRACT

Developers of JavaScript web applications have little tool support for catching errors early in development. In comparison, an abundance of tools exist for statically typed languages, including sophisticated integrated development environments and specialized static analyses. Transferring such technologies to the domain of JavaScript web applications is challenging. In this paper, we discuss the challenges, which include the dynamic aspects of JavaScript and the complex interactions between JavaScript, HTML, and the browser. From this, we present the first static analysis that is capable of reasoning about the flow of control and data in modern JavaScript applications that interact with the HTML DOM and browser API.

One application of such a static analysis is to detect type-related and dataflow-related programming errors. We report on experiments with a range of modern web applications, including Chrome Experiments and IE Test Drive applications, to measure the precision and performance of the technique. The experiments indicate that the analysis is able to show absence of errors related to missing object properties and to identify dead and unreachable code. By measuring the precision of the types inferred for object properties, the analysis is precise enough to show that most expressions have unique types. By also producing precise call graphs, the analysis additionally shows that most invocations in the programs are monomorphic. We furthermore study the usefulness of the analysis to detect spelling errors in the code. Despite the encouraging results, not all problems are solved and some of the experiments indicate a potential for improvement, which allows us to identify central remaining challenges and outline directions for future work.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Algorithms, Verification

[*]Supported by The Danish Research Council for Technology and Production, grant no. 09-064210.

1. INTRODUCTION

A JavaScript web application is in essence an HTML page with JavaScript code and other resources, such as CSS stylesheets and image files. Program execution is driven by events in the user's browser: the page is initially loaded, the user interacts with the mouse and keyboard, timeouts occur, AJAX response messages are received from the server, etc. The event handler code reacts by modifying the program state and the HTML page via its DOM (Document Object Model) and by interacting with the browser API, for example to register new event handlers. Compared to other software platforms, the state of the art in development of such web applications is rather primitive, which makes it difficult to write and maintain robust applications. Statically typed languages, such as Java and C#, have long benefited from advanced IDEs and static analysis techniques with rich capabilities of locating likely programming errors during development. Examples of such tools include Eclipse, Visual Studio, FindBugs, and Klocwork. In contrast, existing tool support for JavaScript web application development is mostly limited to syntax highlighting and primitive code completion in IDEs, such as Eclipse, NetBeans, and Visual Studio, often combined with record/play testing frameworks, such as Selenium, Watir, and Sahi.

The goal of our research is to develop static program analysis techniques that can detect—or show absence of—potential programming errors in JavaScript web applications. We focus on general errors that can be detected without the use of application-specific code annotations. Examples of such errors are (1) dead or unreachable code, which often indicates unintended behavior, (2) calls to built-in functions with a wrong number of arguments or with arguments of unexpected types, and (3) uses of the special JavaScript value `undefined` (which appears when attempting to read a missing object property) at dereferences or at function calls. The existence of the `undefined` value and implicit type coercions in the language means that even minor spelling errors, for example in a property name, often has surprising consequences at runtime. With statically typed languages, the type systems provide a strong foundation for detecting such errors. In contrast, because of the dynamic nature of JavaScript web application code, our analysis must be capable of reasoning about the flow of control and data throughout the applications.

We strive to make the analysis *sound*, meaning that all control flow and dataflow that is possible in the program being analyzed is captured by the analysis such that guarantees can be made about absence of errors. Also, it must be sufficiently *precise* and *fast* such that the user is not overwhelmed with spurious warnings and that the analysis can be integrated into the development cycle.

As an example, Figure 1 shows excerpts from a modern JavaScript web application. If one wants to detect or show absence of errors of the kinds discussed above, a static analysis must reason about the subtle flow of control and data between the JavaScript code, the HTML code, and the browser event system, as explained in the figure text.

TAJS is a program analysis tool for JavaScript [11,12]. To this point, TAJS has been developed to faithfully model the JavaScript language and the core library as specified in the ECMAScript standard [4]. Most real JavaScript programs, however, exist in the context of an HTML page and operate in browsers where they access the HTML DOM and the browser API, which causes considerable challenges to the analysis of the flow of control and data [18]. We now take the step of extending TAJS to also model these aspects of JavaScript web applications.

In summary, the contributions of this paper are the following:

- We discuss the key challenges (Section 2) and suggest an approach toward modeling the JavaScript web application platform in static analysis (Section 4). In particular, this involves considerations about modeling the HTML pages and the event system.
- We show how the TAJS analysis (Section 2.2) can be extended to accommodate for the HTML DOM and the browser API. As result, we obtain the first static analysis tool that is capable of reasoning about the flow of control and data in JavaScript web applications.
- Through experimental evaluation we demonstrate that our model is sufficient to show absence of errors and to detect dead and unreachable code. In addition, we evaluate the precision of the types and call graphs inferred by the analysis (Section 5). We identify strengths and weaknesses of the approaches we have taken and suggest directions for future work (Section 7).

Several program analysis tools and techniques for JavaScript have been developed [1,3,5–11,15,19], however, none of them provide a detailed model of the HTML DOM and the browser API, although all JavaScript web applications utilize those mechanisms. We describe connections to related work in Section 6.

2. CHALLENGES

We begin with a brief tour of the technologies involved and explain the central challenges that exist when developing static analyses for JavaScript web applications. Experienced JavaScript programmers who are used to reasoning "manually" about the behavior of their programs will recognize the issues brought forth here.

2.1 The JavaScript Language

The first obstacle we face is the JavaScript language itself. JavaScript has higher-order functions and closures, exceptions, extensive type coercion rules, and a flexible object model where methods and fields can be added or change types and inheritance relations can be modified during execution. As shown by Richards et al. [18], commonly made assumptions in the research literature about JavaScript programs are often violated by the code actually being written by programmers, and JavaScript is described as "a harsh terrain for static analysis".

Implementations largely follow the ECMAScript standard [4], however, there are subtle deviations. One such example is that many browsers for performance reasons do not implement the specified behavior of deleting properties of the `arguments` object (as in `delete arguments[0]`). Another example is that many browsers for security reasons do not correctly invoke the currently defined `Object` function when constructing objects from literals (as in `x={}`). Other peculiar JavaScript features and incompatibility issues are discussed in the paper on JavaScript semantics by Maffeis et al. [16]. One choice we must make is whether to model the standard or one or more of the existing implementations. We return to this issue in Section 3.

On top of the language, ECMAScript contains a standard library consisting of 161 functions and other objects that all need to be modeled somehow by any tool that analyzes JavaScript web applications. Of particular interest is the `eval` function and its variant `Function` that allow dynamic construction of program code from text strings. Reasoning statically about the behavior of such code obviously requires knowledge about which strings may appear. Even so, studies of how these constructs are used in practice indicate that many cases are amenable to static analysis [14,17,18].

For now, we focus on the 3rd edition of ECMAScript (ECMA-262), which is currently the most widely used version. Supporting the more recent 5th edition requires the analysis to also reason about getters and setters, sealed and frozen objects, stronger reflection capabilities, and the so-called strict mode semantics, in addition to a range of new standard library functions.

2.2 The HTML DOM and Browser API

The browser environment gives rise to additional challenges. The JavaScript representation of HTML documents, CSS properties, and the event system is specified by the W3C DOM standards[2]. The HTML5 specification is currently being developed by the WHATWG group[3]. Together, these specifications contribute additional hundreds of functions and other objects to the program state. It is well known to all web application programmers that browsers do not adhere to these standards. Browsers provide non-standard functionality, and many standard features are not supported[4]. In particular the event systems differ between browsers. Another problem is that no standard exists for the `window` object that acts as the global JavaScript object. Incompatibilities in the underlying JavaScript interpreters mostly involve subtle corner cases in the language, as discussed above, and often go unnoticed by the programmers. In contrast, incompatibilities in the browser environments are a major concern. When developing a program analysis, we need to choose which of these variations to model.

A typical workaround is seen in the following function `addEvent` from the Google Chrome Experiment *Tetris*[5].

[1]`http://www.chromeexperiments.com/detail/js-touch/`

[2]`http://www.w3.org/DOM/`

[3]`http://www.whatwg.org/`

[4]`http://www.quirksmode.org/`

[5]`http://www.chromeexperiments.com/detail/domtris/`

```
 1 <html>
 2 <head>
 3 <script type=''text/javascript''>
 4 window.P3D = {
 5   texture: null,
 6   g: null
 7 };
 8 
 9 P3D.clear = function(f, w, h) {
10   var g = this.g;
11   g.beginPath();
12   g.fillStyle = f;
13   g.fillRect(0, 0, w, h);
14 }
15 
16 function TouchApp() {
17   var _this = this;
18 
19   this.canvas = document.getElementById("cv");
20   P3D.g = this.canvas.getContext("2d");
21   //...
22 
23   this.mViewport = {};
24   this.mViewport.w = 480;
25   this.mViewport.h = 300;
26   //...
27 
28   var tex = new Image();
29   this.ipod.texture = tex;
30   tex.onload = function(){ _this.start(); };
31   tex.src = "20090319144649.png";
32   //...
33 }
34 
35 TouchApp.prototype = {
36   start: function() {
37     //...
38     this.onInterval();
39   },
40 
41   onInterval: function() {
42     //...
43     P3D.clear("#000",
44               this.mViewport.w,
45               this.mViewport.h);
46     //...
47     setTimeout(function(){
48       _this.onInterval();
49     }, 20);
50   }
51   //...
52 }
53 //...
54 </script>
55 </head>
56   <body onload="void( new TouchApp() );">
57     <canvas id="cv" width="480" height="300"/>
58     //...
59   </body>
60 </html>
```

The code at the left is an excerpt from the Google Chrome Experiment *js touch* (where `//...` indicates omitted code). It displays a 3D model of an iPhone and allows the user to interact with it by moving the mouse. The application is written in pure JavaScript and uses the new HTML5 `canvas` object.

Obviously, many things could go wrong when programming such an application. Three examples of correctness properties that the programmer may consider are: (1) Is the parameter `g` on line 11 always an object with a `beginPath` function? If not, a runtime error will occur when that line is executed. (2) In the call to the function `fillRect` on line 13, are the arguments always numeric? If not, the function call will not have the desired effect. (3) Is the function `P3D.clear` on line 9 reachable in some execution? If not, presumably there is an error in the control flow.

To catch such errors – or to show their absence, a static analysis must know about the flow of control and data in the program. In brief, the browser first loads the HTML page and executes the top-level JavaScript code and `load` event handlers. It then executes other event handlers for user input, timeouts, and other events that occur.

In this example application, the code on line 56 in the `onload` attribute of the `body` element creates a new `TouchApp` object and invokes its constructor function defined on line 16. This function looks up the JavaScript DOM object representing the canvas element on line 19 and then stores a reference to its associated `CanvasRenderingContext2D` in the `g` property of the `P3D` object on line 20. Note that `P3D` is a globally available object. Next, on line 28, the constructor function creates a new `Image` object, sets its `load` event handler to the `start` function and finally sets its `src` property. The browser loads the requested image and fires the `load` handler. The `start` function, defined on line 36, does some work and then invokes the `onInterval` function. This function, defined on line 41, calls `P3D.clear` with appropriate arguments taken from the `this.mViewport` object. Finally, using a call to `setTimeout`, it registers itself to be invoked by the browser 20ms later.

By automating this kind of reasoning, a static analysis can detect likely errors in the application code. Analyzing a complex JavaScript program, such as this one, requires a precise model of the JavaScript language, the HTML DOM, and the browser API. For this application, our analysis tool is capable of showing in 9 seconds among many other properties that (1) the variable `g` does always hold an object with a `beginPath` function, (2) the `fillRect` function is always called with numeric arguments, and (3) the function `P3D.clear` is likely to be reachable. In addition, the analysis reports that 98.9% of all property access operations are guaranteed free from `TypeError` exceptions caused by dereferencing `undefined` or `null` and that all calls to browser API functions are given arguments of meaningful types. More statistics for the unabridged experiment is in Section 5.

Figure 1: Excerpts from the Google Chrome Experiment *JS Touch*[1].

```
function addEvent(el, event, handler) {
  if (el.addEventListener)
    el.addEventListener(event, handler, false);
  else if (el.attachEvent)
    el.attachEvent("on" + event, handler);
}
```

Reasoning statically about the behavior of such code requires not only modeling of different browsers but also flow sensitivity (i.e. taking statement order into account) and even path sensitivity (i.e. considering the branch conditions) to see that the calls to `addEventListener` and `attachEvent` do not cause `TypeError` exceptions.

Since all execution is driven by events, the analysis must also model the event system, which includes the dynamic registration and removal of different kinds of event handlers, as in the `addEvent` function above, the event bubbling and capturing mechanism, and the event object properties that depend on the specific kind of event. The event handlers work as callbacks, which often leads to fragmented code with unclear flow of control that the static analysis must resolve. A small example is seen in the Chrome Experiment *Aquarium*[6] (abbreviated for presentation):

[6]`http://www.chromeexperiments.com/detail/aquarium/`

```
function mmouse(event) {
  mousex=event.pageX;
  mousey=event.pageY;
}
function work() {
  var dx=mousex-pesti[x].x;
  var dy=mousey-pesti[x].y;
  //...
}
setInterval(work,10);
```

The function `mmouse`, which is elsewhere registered as an event handler, stores information about the event in two global variables, `mousex` and `mousey`, that are read in another event handler, `work`. Unless these two variables are properly initialized, `dx` and `dy` will get the special value `NaN` if the `work` function happens to be triggered before `mmouse`, which will likely result in an error later in the execution.

For event handlers defined as HTML attributes, the HTML document structure interferes with the execution scope chains that are used when resolving variables. If an event handler defined literally as an attribute in an HTML element is triggered, the scope chain includes all the DOM objects that make up the path from the HTML element to the root of the document. This means that dataflow in the JavaScript code in general cannot be analyzed separately from the HTML code. The following example illustrates this mechanism:

```
<script type=''text/javascript''>
  var src = "foo.png";
</script>
<img src="bar.png" onclick="alert(src)"/>
```

The value of `src` inside the `onclick` event handler is that of the `src` attribute of the `img` element, not `foo.png` as one might have expected.

Many properties in the ECMAScript native objects have special attributes, such as *ReadOnly*, which also must be accounted for unless sacrificing either soundness or precision. Likewise, many DOM objects behave differently from ordinary objects. As an example, a new `form` element is created with `document.createElement('form')`, not with `new HTMLFormElement` although all `form` elements inherit from `HTMLFormElement.prototype`.

Besides the extent and the variations of browser environments, other concerns when developing a static analysis tool relate to the prevalence of nontrivial built-in setters, that is, assignment operations that involve complex conversions or other side-effects. For example, writing to the `onclick` property of an HTML element object causes a string to be treated as event handler code. Another example is the use of *value correspondence* where HTML element attributes are represented in multiple JavaScript objects. For instance, the `src` attribute value of an `img` element appears both directly as a property of the `img` element object and indirectly as a property of an object that can be reached via the `attributes` property of the `img` element object. These are essentially aliases (although the former is always an absolute URL even when the latter is a relative URL), and modifications to one also affect the other, much like the connection between ordinary JavaScript function parameters and the `arguments` object. Consider also the `window.location` property, which holds a `Location` object. Assigning a new URL string to this property causes the browser to go to that URL after the current event handler and various unload handlers have been executed. As yet another example, writing a string to the (also nonstandard but widely used) `innerHTML` property of an element object causes the string to be parsed as HTML and converted to a DOM object structure, which then replaces the element contents.

A related issue is the *element lookup* mechanism, which provides support for `getElementById` and related functions. If an element with an `id` attribute is inserted into the HTML document, it is automatically added to the browser's element ID table for quick lookup. Similarly, `documents.images` automatically contains references to all images in the current HTML document.

2.3 Application Development Practice

Further complications are introduced by common application development practice. Although JavaScript is an interpreted language (perhaps with JIT compilation, transparently to the programmer) in practice it makes sense to distinguish between "source code" and "executable code". The reason is that JavaScript web application code is often subjected to *minification* (and sometimes also *obfuscation*) to reduce the code size and thereby make the applications load faster. A related trick is *lazy loading* where the applications are divided into parts that are loaded incrementally using AJAX or dynamically constructed `script` elements.

An example of lazy loading using a dynamically created `script` tag occurs in the Google Analytics[7] tool for collecting visitor statistics:

```
<script type="text/javascript">
  (function() {
    var ga = document.createElement('script');
    ga.type = 'text/javascript';
    ga.async = true;
    ga.src =
      ('https:' == document.location.protocol ?
       'https://ssl' :
       'http://www') + '.google-analytics.com/ga.js';
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(ga, s);
  })();
</script>
```

Since our aim is to develop an analysis tool that can help the programmers catch errors during development, we choose to focus on the source code stage, as the programmers see the application before these techniques are applied. This means that we in many cases sidestep the issue of analyzing dynamically generated code. It also means, however, that the analysis tool we develop is not designed to be used for all the JavaScript web application code that is immediately available on public web sites, such as Gmail or Office Web Apps.

Many applications build on libraries that alleviate browser incompatibility problems, provide class-like abstractions and advanced GUI widgets and effects, and simplify common tasks, such as navigation in the HTML DOM structures and AJAX communication. This includes general libraries, for example jQuery, MooTools, and Prototype, but also a myriad of more specialized libraries, such as plugins for jQuery. From a static analysis point of view, libraries such as these in many cases make it difficult to track flow of control and data. By providing their own abstractions on top of event handling and DOM objects, a high degree of context sensitivity and detailed modeling of heap structures may be required by the analysis. An example of a challenging library construct is the `$` function in jQuery, which has very different behavior depending on whether it is passed a function, an HTML string, a CSS string, or a DOM element.

[7] `http://www.google.com/analytics/`

3. THE TAJS ANALYZER

We base the current work on the TAJS analysis tool that is described in previous publications [11,12]. TAJS is a whole-program flow analysis that supports the full JavaScript language as defined in the ECMA-262 specification [4], including the entire standard library except `eval`. The analysis is designed to be sound (although working with a real-world language and having no standardized formal semantics of the language nor of the HTML DOM and browser API, soundness is not formally proven). To this point, we do not consider the deviations from the ECMAScript standard that are discussed in Section 2.1, the reason being that these deviations are mostly corner cases that are irrelevant to most applications we have studied. If the need should arise, for all the deviations we are aware of, it is only a matter of making minor adjustments to the analysis tool.

TAJS is based on the classic monotone framework [13] using a highly specialized analysis lattice structure. The lattice is based on constant propagation for all the possible primitive types of JavaScript values. In addition, the lattice includes call graph information, allowing on-the-fly construction of the call graph to handle higher-order functions. It also contains a model of the heap based on allocation site abstraction extended with recency abstraction [2].

The analysis is object sensitive, meaning that it distinguishes between calling contexts with different values of `this`. It is also flow sensitive, meaning that it distinguishes between different program points (maintaining separate abstract states for different program points), and it has a simple form of path sensitivity to distinguish between different branches of conditionals.

On top of this, lazy propagation is used to ensure that only relevant parts of the abstract states are propagated, which improves both performance and precision [12].

Altogether, this foundation largely addresses the challenges that are directly related to the ECMAScript language specification.

4. MODELING THE HTML DOM AND BROWSER API

We now present our approach to extending the analysis to accommodate for the HTML DOM and the browser API.

Regarding the multitude of APIs supported by different browsers that exist, we choose to model the parts that we believe is most widely used: the DOM Core, DOM HTML, and DOM Events modules of the W3C recommendations (Level 2, plus selected parts of Level 3), the essential parts of `window`[8] and related nonstandard objects, and the `canvas` and related objects from WHATWG's HTML5 (as of January 2011). The latter allows us to test the analysis on web applications that exploit cutting edge functionality supported by the newest browsers.

In total, the extensions comprise around 250 abstract objects with 500 properties and 200 transfer functions. To give an impression of the complexity, Figure 2 shows a small part of the object hierarchy of the initial abstract state. Each node represents an abstract object with its associated properties and functions, and the edges represent internal prototype links. The symbols `@` and `*` in the names indicate whether the abstract objects represent single or multiple concrete objects.

[8]`https://developer.mozilla.org/en/DOM/window`

4.1 HTML Objects

The HTML page and resources linked to from the page define not only the program code but also the initial state for the execution, including the HTML document object structure, element lookup tables, and event handlers.

At runtime, each HTML element gives rise to a range of JavaScript objects, and new HTML elements can be created dynamically. We need a bounded representation to ensure that the program analysis terminates (technically, the analysis lattice must have finite height), thus abstraction is necessary. A simple approach is to represent all HTML objects as one abstract object. This is essentially what is done in other program analyses [7, 8] that perform a less detailed analysis than what we aim for. To preserve the inheritance relationships between the DOM objects, we choose an abstraction where all constructor objects and prototype objects are kept separate and that distinguishes between HTML elements of different kinds but where multiple elements of the same kind are merged. As an example, the `HTMLInputElement` abstract object (see Figure 2) models all HTML `input` elements. It has properties such as `accessKey` and `checked`, which in the analysis have types `String` and `Boolean`, respectively. The abstract object inherits from `HTMLInputElement.prototype`. This object contains common functionality, such as the `focus` function, shared by all `HTMLInputElement` objects. Looking further up the prototype chain we find `HTMLElement.prototype`, `Element.prototype` and finally `Node.prototype`, which define shared functionality of increasingly general character. Other types of HTML elements, such as `form` or `canvas` elements are similarly modeled by separate abstract objects. This approach respects the inheritance relationships and it smoothly handles programs that dynamically modify the central DOM objects, for example by adding new methods to the prototype objects.

To model the element lookup mechanism (see Section 2.2), we extend TAJS's notion of abstract states with appropriate maps, e.g. from element IDs to sets of abstract objects. The initial abstract state is populated with the IDs that occur in the HTML page. If the HTML page contains an `input` element with an attribute `id="foo"` then the ID map in the abstract state maps `foo` to the `HTMLInputElement` abstract object. These maps are updated during the dataflow analysis if new `id` attributes are inserted into the page. As result, `getElementById` and related functions are modeled soundly and with reasonable precision.

4.2 Events

As discussed in Section 2.2, the analysis must be extended to model dynamic registration, triggering, and removal of event handlers. This can be done with various levels of precision. We describe our choices in the following and evaluate the resulting system in Section 5.

First, we extend TAJS's abstract states again, this time with a collection of set of references to abstract objects that model the event handler function objects. To distinguish between different kinds of events and event objects, we maintain one such set for each of the following categories of events: *load*, *mouse*, *keyboard*, *timeout*, *ajax*, and *other*. Object references are added to these sets either statically, due to presence of event attributes (`onload`, `onclick`, etc.) in the HTML page, or dynamically when encountering calls to `addEventListener` or assignments to event attributes during

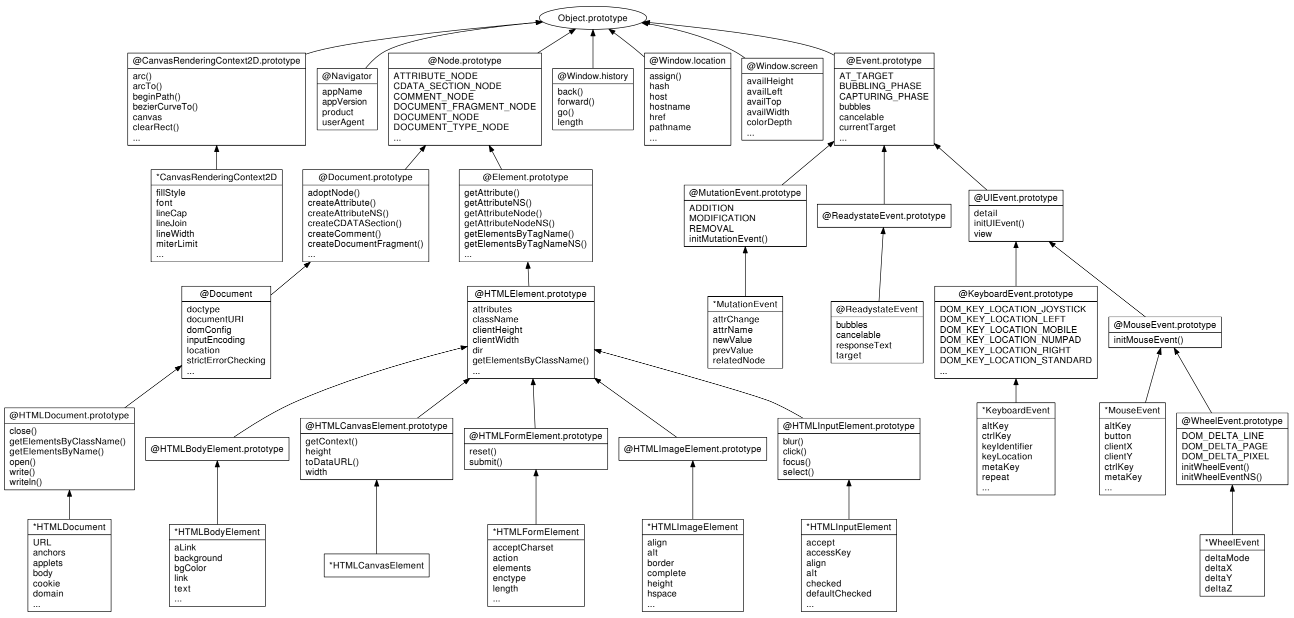


Figure 2: An excerpt from the HTML object hierarchy.

the analysis. This means that the abstract states always contain an upper approximation of which event handlers exist. Note that we choose to abstract away the information about where in the HTML DOM tree the event handlers are registered (i.e. the `currentTarget` of the events). This allows us to ignore event bubbling and capturing. Similarly, we ignore removal of event handlers (`removeEventListener`). These choices may of course affect precision, but analysis soundness is preserved.

Next, we need to model how events are triggered. A JavaScript web application is executed by first running the top-level code and then, until the page is unloaded, running event handlers as reaction to events. Each event handler is executed until completion, without being interrupted when new events occur.

In TAJS, JavaScript program code is represented by flow graphs, which are graphs where nodes correspond to primitive instructions and edges correspond to control flow (see [11]). We have considered different approaches to incorporating the event handler execution loop after the top-level code in the flow graph:

- As a single loop where all event handlers in the current abstract state are executed non-deterministically. This is a simple and sound approach, but it does not maintain the order of execution of the individual event handlers.
- Using a state machine to model the currently registered event handlers. This is a considerably more complex approach, but it can in principle more precisely keep track of the possible order of execution of the event handlers.

Through preliminary experiments we have found for the correctness properties that we focus on, the execution order of event handlers is often not crucial for the analysis precision. However, we found that it is important to model the fact that *load* handlers are executed before the other kinds of event handlers. For this reason, we model the execution of event handlers as shown in Figure 3. (To simplify the illustration we here ignore flow of runtime exceptions.) The flow graph for the top-level JavaScript is extended to include two non-deterministic event loops, first one for the *load* event handlers and then one for the other kinds.

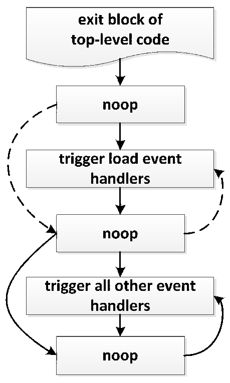

exit block of top-level code
noop
trigger load event handlers
noop
trigger all other event handlers
noop

Figure 3: Modeling events in the flow graphs.

If only a single *load* handler is registered (and it is not subsequently removed) then we know that it is definitely executed once, and thus we can effectively remove the dashed edges. This increases precision because otherwise all state initialized by *load* handlers would be modeled as maybe absent.

When triggering event handlers, we exploit the fact that the abstract states distinguish between the different event categories listed above. This allows us to model the event objects appropriately, for example using the abstract object `KeyboardEvent` (see Figure 2) to model keyboard event objects. Moreover, the analysis abstraction used in TAJS already has a fine-grained model of scope chains, so it is relatively easy to incorporate the HTML element objects to take the issues regarding scope chains (see Section 2.2) into account.

4.3 Special Object Properties

As discussed in Section 2.2, writes to certain object properties, such as `onclick`, `src`, and `innerHTML`, have special side-effects. The TAJS analysis infrastructure conveniently supports specialized transfer functions for such operations. This allows us to trigger the necessary modifications of the abstract state when property write operations occur for certain combinations of abstract objects and property names. With this, we can easily handle code such as the following that dynamically constructs an `img` element and sets the `id` and `onclick` properties, which affects not only the `img` object itself but also the element ID lookup map and the event handler set:

```
var i = document.createElement("img");
f.id = "myImage";
f.onclick = function {...}
```

With this approach, the abstractions made elsewhere in the analysis can in principle lead to a cascade of spurious warnings. If the analysis detects a property write operation that involves one of the relevant objects but where the property name is unknown due to abstraction, a fully sound analysis would be required to trigger all the possible specialized transfer functions, which could cause a considerable loss of analysis precision. Instead, if this situation occurs, we choose to sacrifice soundness such that the analysis simply emits a general warning and skips the modeling of the special side-effects for that particular property write operation. In our experiments (see Section 5), this occurs 0 times, indicating that the analysis is generally precise enough to avoid the problem.

4.4 Dynamically Generated Code

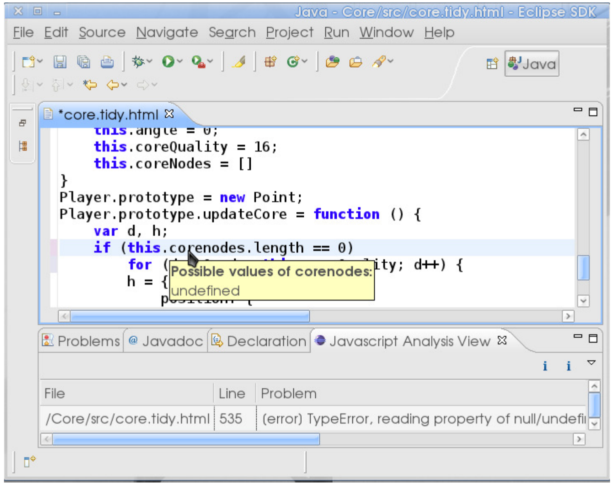
We extend TAJS to support certain common cases involving `eval` and the related functions `Function`, `setTimeout`, and `setInterval`. Programmers who are not familiar with higher-order functions often simulate them by using strings instead, such as in this example from the program *Fractal Landscape*[9]:

```
animInterval = setInterval("animatedDraw()", 100);
```

This code works because the function `setInterval` supports being called with a string that will get evaluated in the global scope at the specified intervals. To accommodate for this, TAJS recognizes the syntax of a string consisting of a simple function call. The analysis transfer function for `setInterval` collects not only function objects but also such strings that represent event handler functions. When modeling the triggering of event handlers, the latter functions are then looked up in the global scope.

An often used application of `eval` is to parse JSON data received using AJAX. JSON data describes simple JavaScript object structures that cannot contain functions. TAJS can be configured to assume that string values that are read from AJAX connections contain only JSON data. We model this with the special dataflow value JSONString. If this abstract value is passed to `eval`, the analysis knows that no side-effects can happen, so the result can be modeled using an abstract value consisting of a generic abstract object and unknown primitive values.

[9]`http://10k.aneventapart.com/Entry/60`


Java - Core/src/core.tidy.html - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
*core.tidy.html
Problems
Javadoc
Declaration
Javascript Analysis View

```
        this.angle = 0;
        this.coreQuality = 16;
        this.coreNodes = []
}
Player.prototype = new Point;
Player.prototype.updateCore = function () {
    var d, h;
    if (this.corenodes.length == 0)
        for (                    ity; d++) {
        h = {
            position {
```

Possible values of corenodes:
undefined

File	Line	Problem
/Core/src/core.tidy.html	535	(error) TypeError, reading property of null/undefi

Figure 4: The TAJS analysis plug-in for Eclipse, reporting a programming error and highlighting the type inferred for the selected expression.

5. EVALUATION

We have extended the pre-existing TAJS analysis tool according to Section 4. The tool is implemented in Java and uses the JavaScript parser from the Mozilla Rhino project[10]. The new extensions amount to 7,500 lines of code on top of the existing 21,000 lines (excluding Rhino). Separately, the analysis is integrated into the Eclipse IDE as a plug-in that allows the programmer to view various aspects of the analysis results, as demonstrated in Figure 4.

5.1 Research Questions

With the implementation, we consider the following research questions regarding the quality of the analysis:

Q1 We wish to study the ability of the tool to detect programming errors of the kinds discussed in Section 1. Given that we do not expect many errors in the benchmark programs that presumable are thoroughly tested already, one way to study the analysis precision is to ask: To what extent can the analysis show the absence of errors in real programs? Since the analysis is designed to be sound (however see Section 4.3), absence of a warning from the tool can be interpreted as absence of an error in the program being analyzed.

Q2 For programs with errors (again, of the kinds discussed in Section 1), can the analysis help the programmer find the errors? Specifically, are the warning messages produced by the tool useful toward leading the programmer to the source of the errors?

Q3 Having a good approximation of the call graph of a program is a foundation for other potential applications, such as program comprehension or optimization. This leads to the question: How precise is the call graph inferred by the analysis?

Q4 Similarly to the previous question, how precise are the inferred types?

Q5 Does the analysis succeed in identifying dead or unreachable code? In some situations, dead or unreach-

[10]`http://www.mozilla.org/rhino`

able code is unintended by the programmer and hence indicates errors. The ability of the analysis tool to detect such code can in principle also be used to reduce application code size before deployment.

5.2 Benchmark Programs

Our benchmark programs are drawn from three different sources: *Chrome Experiments*[11], *Internet Explorer 9 Test Drive*[12] and the *10K Apart Challenge*[13]. Chrome Experiments consist of JavaScript web applications that demonstrate the JavaScript features of the Chrome browser. Despite the name, the majority of these applications can be executed in any modern browser. Most of the applications use the new HTML5 `canvas` element to create graphics in various ways including games and simulations. Internet Explorer 9 Test Drive is a collection of applications written to test and demonstrate features of the newest version of the Internet Explorer browser. We exclude applications that contain no or very little JavaScript code or rely on Flash or other browser plug-ins. The 10K Apart Challenge collection consists of JavaScript web applications that are less than 10KB in size including code and markup.

The programmers of some of the 10K Apart Challenge applications have applied `eval` creatively to reduce the code size in ways that we believe are not representative of ordinary JavaScript web applications. For this reason, we disregard applications that syntactically use `eval` in other ways than those covered in Section 4.4. Moreover, analyzing applications that involve large libraries, such as jQuery, MooTools, and Prototype, is particularly challenging for the reasons discussed in Section 2.3. At present, we limit our level of ambition to applications that do not depend on such libraries. The applications we thereby exclude can form an interesting basis for future work on static analysis in relation to `eval` or libraries.

The resulting collection of 53 JavaScript web applications is listed in Table 1 and available at `http://www.brics.dk/TAJS/dom-benchmarks`. In the table, the columns LOC, BB, and Time show the number of lines of code (pretty-printed and including HTML), the number of basic blocks of JavaScript code, and the analysis time (running on a 2.53Ghz Mac OS X computer with 4GB of memory). Dynamically generated code of the kind discussed in Section 4.4 appears in 17% of the applications. All the applications involve HTML and the event system, so none of them could be analyzed with TAJS before the new extensions described in this paper.

5.3 Experiments and Results

We address each research question, Q1–Q5, in turn with experiments and evaluation.

For Q1, we focus on the following kinds of likely errors:

- Invoking a non-function value as a function.
- Accessing a property of the special values `undefined` or `null`.
- Reading an absent object property using the fixed-property notation (we here ignore operations that use the notation for dynamically computed property names).

[11]`http://www.chromeexperiments.com/`
[12]`http://ie.microsoft.com/testdrive/`
[13]`http://10k.aneventapart.com/`

The first two cause `TypeError` exceptions; the third yields the value `undefined`. Technically, these situations are not necessarily errors, but they are rarely intended by the programmer. One exception is that absent properties may appear in browser feature detection code, in which case the analysis can help ensuring that the code works for the browser being modeled.

For each error category we measure the percentage of flow graph nodes for which TAJS decides not to issue a warning of the particular kind. The results are shown in the three columns labelled CF, PA and FPU in Table 1, corresponding to the three kinds of likely errors. We see that TAJS is able to show absence of these particular kinds of errors for most of the program code, in many cases more than 90% of the places in the code where the errors could potentially occur. There are a few outliers that get lower results: Both *Tetris* and *Minesweeper* rely on multi-dimensional arrays for most of their state, which leads to imprecision in property reads. Complex object models, such as in the *Raytracer* benchmark, are also the cause of some imprecision.

As we do not expect our benchmarks to contain any of the error conditions listed above, we answer Q2 by introducing errors into the benchmark programs at random. We simulate spelling errors made by the programmer by picking a random read or write property operation that uses the fixed-property notation (i.e. the `.` operator) and replacing the property name with a different one. For each benchmark, we run the analysis repeatedly and manually inspect whether each spelling error results in a warning by the analysis tool and how "useful" this warning is. We measure usefulness by two criteria: the source location of the warning that is issued should be close to where the error is inserted, and the warning should be prominent, i.e. appear near the top in the list of analysis messages.

This process has been carried out for a random subset of our benchmark programs. All show a common pattern: Spelling errors at read operations are reliably detected with a warning that appears at the top of the list of analysis messages. Not surprisingly, spelling errors introduced at write operations have more diverse consequences, as any warning will only occur when the program later attempts to read the property that was affected. Furthermore, errors introduced in connection to side-effects that are not modeled by TAJS, such as the DOM property `style`, are often not detected.

We present the results for the *Mr. Potato Gun* benchmark as a representative example. We analyzed it 50 times with a different spelling error introduced each time. In 84% of the cases the error resulted in one or more warnings. Of the errors introduced, 7 were in write operations and 43 in read operations. Only one of the write operation errors was detected, resulting in the warning `ReferenceError, reading absent property: (computed name)`, which is a high-priority warning that is issued for the location where the program tries to read the property that was misspelled. For the read operations, each error was reported as a warning such as `ReferenceError, reading absent property: AQ` issued for the exact source location of the error.

These experiments indicate that the information obtained by the analysis can be useful for detecting spelling errors in the program code, but a more thorough investigation is necessary to give a solid answer to Q2.

For Q3 we wish to evaluate the precision of the computed call graph. This is measured by calculating the ratio of call

	LOC	BB	CF	PA	FPU	UF	DC	MC	ATS	Time
3D Demo	1205	1770	99.2	97.9	98.9	125/58	7	100.0%	1.1	8.0s
Another World	1477	1437	100.0	99.3	98.3	45/0	0	100.0%	1.3	20.7s
Apophis	1140	1319	100.0	80.4	80.4	58/0	0	100.0%	1.1	16.3s
Aquarium	166	151	93.7	87.6	72.8	9/0	0	100.0%	1.3	3.2s
Bing-Bong	1148	1176	100.0	87.9	92.5	66/0	2	100.0%	1.1	17.9s
Blob	596	748	100.0	95.6	97.4	37/2	19	100.0%	1.0	6.4s
Bomomo	2905	3885	80.6	96.3	61.2	170/8	10	100.0%	1.3	57.1s
Breathing Galaxies	101	101	94.7	100.0	91.3	5/0	0	100.0%	1.0	1.3s
Browser Ball	434	771	99.0	97.7	98.1	32/11	0	100.0%	1.0	4.2s
Burn Canvas	180	207	100.0	97.7	100.0	12/0	0	100.0%	1.1	0.9s
Catch It	207	200	97.2	86.0	98.6	11/0	0	100.0%	1.1	3.3s
Core	566	611	100.0	98.7	98.4	23/1	10	100.0%	1.0	5.6s
JS Touch	1452	762	100.0	98.9	98.1	48/8	9	100.0%	1.1	5.8s
Kaleidoscope	249	334	98.9	88.6	82.1	14/1	3	100.0%	1.1	6.1s
Keylight	731	791	99.4	96.1	98.7	37/0	24	100.0%	1.0	7.4s
Liquid Particles	253	205	100.0	98.5	100.0	11/4	2	100.0%	1.0	1.8s
Magnetic	415	339	100.0	95.5	100.0	19/0	1	100.0%	1.0	4.1s
Orange Tunnel	102	133	100.0	80.3	100.0	7/1	0	100.0%	1.1	2.6s
Plane Deformations	552	514	100.0	100.0	95.1	17/0	5	100.0%	1.5	1.5s
Plasma	204	228	100.0	100.0	100.0	9/0	2	100.0%	1.1	1.6s
Raytracer	1380	1515	87.2	93.7	55.5	78/24	33	90.1%	1.3	20.6s
Starfield	231	393	98.7	79.0	87.6	21/6	2	100.0%	1.2	2.9s
Tetris	827	803	95.1	79.6	58.8	39/4	2	100.0%	1.8	9.7s
Trail	212	166	100.0	98.0	98.2	10/0	0	100.0%	1.0	12s
Voronoi	525	1066	100.0	78.8	99.7	70/7	10	99.5%	1.1	10.5s
Water Type	309	266	100.0	95.0	97.2	14/0	0	100.0%	1.1	1.9s
Asteroid Belt	319	707	100.0	94.6	97.0	27/5	30	100.0%	1.1	3.1s
Browser Flip	507	324	100.0	88.5	97.6	10/0	1	100.0%	1.1	3.2s
FishIE	336	717	99.4	96.0	95.5	19/2	30	100.0%	1.0	3.3s
Flying Images	589	497	100.0	97.5	91.8	33/0	0	100.0%	1.0	3.9s
Mr. Potato Gun	817	1015	98.7	97.6	95.0	31/1	12	100.0%	1.1	7.8s
10k World	439	930	100.0	86.9	91.4	47/2	3	100.0%	1.1	15.1s
3D Maker	427	773	100.0	67.3	70.5	29/3	0	100.0%	1.2	10.3s
Attractor	445	696	97.0	92.3	91.2	34/0	1	100.0%	1.3	5.8s
Defend Yourself	517	601	94.7	78.6	90.1	31/0	0	100.0%	1.1	7.9s
Earth Night Lights	129	245	100.0	100.0	100.0	14/0	0	100.0%	1.0	1.1s
Filterrific	697	995	96.5	86.7	72.3	55/0	4	99.0%	1.2	29.8s
Flatwar	444	685	99.2	97.4	93.6	19/1	0	100.0%	1.1	6.9s
Floating Bubbles	381	693	100.0	89.9	99.7	39/6	23	100.0%	1.1	6.4s
Fractal Landscape	171	162	100.0	100.0	97.7	7/0	0	100.0%	1.0	0.8s
Gravity	231	258	98.7	87.3	90.9	9/0	0	100.0%	1.0	5.2s
Heatmap	255	350	95.1	93.6	87.3	30/1	2	97.3%	1.1	3.1s
Last Man Standing	300	570	100.0	95.9	100.0	33/1	2	100.0%	1.1	4.2s
Lines	459	931	97.3	88.5	93.9	22/6	2	100.0%	1.2	4.7s
Minesweeper	175	358	100.0	81.4	68.5	15/0	3	100.0%	1.3	4.7s
NBody	479	450	99.1	68.7	43.6	15/0	0	100.0%	1.6	50.8s
RGB Color Wheel	455	700	97.7	82.7	85.0	38/0	2	100.0%	1.1	5.6s
Sinuous	349	488	100.0	96.3	98.5	23/0	10	100.0%	1.0	5.5s
Snowpar	338	519	100.0	88.6	88.6	31/0	0	100.0%	1.2	3.2s
Stairs to Heaven	210	422	100.0	94.5	100.0	25/8	1	100.0%	1.0	2.5s
Sudoku	316	612	96.2	81.0	60.4	33/0	0	100.0%	1.3	12.1s
TicTacToe	304	590	100.0	74.0	100.0	19/0	0	100.0%	1.2	7.4s
Zmeyko	344	601	100.0	96.7	96.3	33/1	0	100.0%	1.0	7.0s

Table 1: Benchmark results for Chrome Experiments, IE Test Drive and 10K Apart Challenge applications. The columns from left to right are: lines of code (LOC), number of basic blocks (BB), percentage of call site operations shown to invoke a function value (CF), property read operations where the base object is shown to be non-`null` and non-`undefined` (PA), fixed-property read operations not resulting in `undefined` (FPU), number of functions in total / number of functions shown to be definitely unreachable (UF), number of dead code operations (DC), percentage of call sites that are shown to be monomorphic (MC), average type size for all property read operations (ATS), and analysis time (Time).

sites with a single invocation target compared to the total number of call sites in the program. If this ratio is one then every call site is monomorphic, i.e. it has a single invocation target. If a call site has a non-function value as a potential invocation target this is not included in the number of targets, since such a value would always result in a runtime error. This measure can be seen in the MC column. In Table 1 we see that despite the fact that JavaScript supports both the prototype lookup mechanism and higher-order functions, the analysis is able to show for 49 of the 53 of the benchmark programs that all call sites have a single invocation target, which gives testimony to the high precision of the analysis.

For Q4 we wish to measure the precision of the computed types. The analysis tracks values of the following types: *boolean*, *number*, *string*, *object* (including null and function values) and the special type *undefined*. This means that an object property could potentially hold values of up to five different types. We measure this aspect of the accuracy of the analysis by calculating the average number of different types for all property read operations in the given program (excluding operations that the analysis finds to be unreachable). If this number is 1 then every read operation results in values of a unique type on all possible executions. The ATS column in Table 1 shows the resulting numbers. Despite the fact that the types of object properties may change dynamically in JavaScript, we note that the analysis is precise enough to show that the average number of different types for each property read operation in these benchmarks is quite close to 1. Of the 26,870 property read operations that appear in the benchmarks, the analysis finds that at most 4,019 can have multiple types.

For the last research question, Q5, we measure both unreachable code and dead code. Unreachable code consists of operations (i.e. flow graph nodes) that are never executed, and dead code is defined to be reachable assignments to properties that are never read. Write operations to special DOM properties, such as `onload`, may have side-effects, so even if there are no corresponding read operations in the program we do not count them as dead code.

The column labelled UF in Table 1 contains the total number of function in the program and how many of them are determined by TAJS to be unreachable. Some of the benchmarks use third-party libraries that are inlined directly in the source code, which explains the large number of unreachable functions in some benchmarks, such as *3D Demo* and *Raytracer*. All code that is found to be unreachable can safely be removed (unless the analysis detects the special situation discussed in Section 4.3), which would significantly reduce code size in some cases. Most current minifiers either unsoundly remove all functions not referenced syntactically in the code or simply do not remove any functions at all. With static analysis, guaranteed behavior preserving minification becomes possible.

The column labelled DC lists the number of dead code operations in each program. We see that the analysis is capable of locating many instances of dead code. Most of the dead code being detected appears to be code left from earlier revisions of the programs. For example, in the *Keylight* benchmark, a flag named `mouseIsDown` is set in all event handlers but it is never read.

The main threat to validity of our conclusions is that our benchmarks may not be representative for typical JavaScript web applications. For the reasons described in Section 5.2 we have excluded applications that rely on large libraries or on complex dynamically generated code. We will focus our attention on these two remaining challenges in future work. Nevertheless, the benchmarks we consider are written by many different programmers, they exhibit a large variety of the functionality supported by the HTML DOM and the browser API, and our experiments show that the program analysis is able to infer many nontrivial properties about their behavior.

6. RELATED WORK

Previous work on static analysis of JavaScript code has focused on the language itself, and often for restricted subsets of the language. To the best of our knowledge, the work reported on in this paper is the first that also models the nontrivial connections between the HTML page and the program code in JavaScript web applications.

One of the first attempts at developing static analysis for JavaScript was done by Anderson et al. who developed a type system and inference algorithm for modeling definite presence and potential absence of object properties in a small subset of JavaScript [1]. The abstract domain used in TAJS subsumes such information. Other early work includes Thiemann's type system [19]. It has a soundness proof but no implementation. Although not tied to JavaScript in particular, Thiemann has also designed a type system for catching errors related to manipulation of DOM structures, in particular to ensure that no loops occur [20].

More recently, Jang and Choe have presented a points-to analysis for a restricted subset of JavaScript based on set constraints [10]. The points-to results are used for optimizations that inline property accesses. In comparison, our analysis yields points-to information as part of the result and supports more features of the language.

The Gatekeeper project by Guarnieri and Livshits includes an Andersen-style points-to analysis for JavaScript [6, 7]. The results of the analysis are used for verifying custom security policies expressed in datalog. The analysis uses a mock-up of the DOM API written in JavaScript and essentially ignores the HTML constituents.

Perhaps most closely related to our work is that of Guha et al. who use a k-CFA analysis to extract a model of the client behavior in an AJAX application as seen from the server [8]. Their paper briefly discusses some of the challenges that relate to events, dynamically generated code, and libraries, but the focus of the paper is on the application for building intrusion-preventing proxies. In comparison, our analysis has a more precise treatment of dataflow and event handlers in connection to the DOM.

Recent work by Guha et al. considers a combination of a type system and a flow analysis to reason about uses of the `typeof` operator in JavaScript code with type annotations [9]. The `typeof` operator appears in 11 of our 53 benchmarks, and TAJS models it with a special transfer function.

Chugh et al. use staged information flow analysis to protect against dynamic loading of malicious code [3]. The analysis identifies fields that can flow into dynamically loaded code and creates runtime monitors to ensure that they are not accessed from untrusted code. The analysis uses a coarse abstraction of the HTML page and the browser API, without considering the challenges we describe in Section 2.

Logozzo and Venter's RATA analysis uses light-weight abstract interpretation to specialize the general JavaScript number type to integer and floating point types for optimization purposes [15]. Making this distinction in the abstract domain used in TAJS would be a straightforward task.

One way to guide the design of an analysis is to survey the practical use of the language. In one such survey by Richards et al., it is shown that many of the dynamic features of JavaScript are not widely used in practice [18]. The study shows that the majority of method invocations in JavaScript are monomorphic. Our experimental results confirm this observation, but using practically sound static analysis instead of runtime measurements. In later work, the use of `eval` is studied [17]. The authors show that the categories of `eval` that are now supported by TAJS, i.e. JSON data and simple function calls, are often used. It is also shown that `eval` is used for lazy loading and as artifacts of generated code, which, as discussed in Section 2.3, is outside the scope of TAJS.

7. CONCLUSION

We have presented the first static analysis that is capable of reasoning precisely about the control flow and dataflow in JavaScript applications that run in a browser environment. The analysis has been implemented as an extension of the TAJS tool and models both the DOM model of the HTML page and browser API. This includes the HTML element object hierarchy and the event-driven execution model. In the process we have identified the key areas where modeling the browser is important for precision and challenging for static analysis.

Our experimental evaluation of the performance of the analysis indicates that (1) the analysis is able to show absence of common programming errors in the benchmark programs, (2) the analysis can help detecting potential errors, such as misspelled property names, (3) the computed call graphs are precise as most call sites are shown to be monomorphic, (4) the computed types are precise as many expressions are shown to have unique types, and (5) the analysis is able to identify dead code and unreachable functions. Such information can give a foundation for providing better tool support for JavaScript web application developers.

Interesting challenges remain. First, more work is required for investigating the more complicated uses of dynamically generated code. Second, better techniques are needed to handle commonly used libraries. Third, the techniques presented here can be adapted to model other JavaScript environments, such as desktop widgets or browser extensions.

8. REFERENCES

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.

[2] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, June 2009.

[4] ECMA. ECMAScript Language Specification, 3rd edition. ECMA-262.

[5] S. Fink and J. Dolby. WALA – The T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`.

[6] S. Guarnieri and B. Livshits. Gulfstream: Staged static analysis for streaming JavaScript applications. In *Proc. USENIX Conference on Web Application Development, WebApps '10*, June 2010.

[7] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. 18th USENIX Security Symposium, Security '09*, August 2009.

[8] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. 18th International Conference on World Wide Web, WWW '09*, May 2009.

[9] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. Programming Languages and Systems, 20th European Symposium on Programming, ESOP '11*, LNCS. Springer-Verlag, March/April 2011.

[10] D. Jang and K.-M. Choe. Points-to analysis for JavaScript. In *Proc. 24th Annual ACM Symposium on Applied Computing, SAC '09, Programming Language Track*, March 2009.

[11] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*, pages 238–255. Springer-Verlag, August 2009.

[12] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium, SAS '10*, volume 6337 of *LNCS*, pages 238–256. Springer-Verlag, September 2010.

[13] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. Springer-Verlag.

[14] R. Kromann-Larsen and R. Simonsen. Statisk analyse af JavaScript: Indledende arbejde. Master's thesis, Department of Computer Science, Aarhus University, 2007. (In Danish).

[15] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *Proc. 19th International Conference on Compiler Construction, CC '10*, volume 6011 of *LNCS*. Springer-Verlag, March 2010.

[16] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. 6th Asian Symposium on Programming Languages and Systems, APLAS '08*, volume 5356 of *LNCS*. Springer-Verlag, December 2008.

[17] 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 *Proc. 25th European Conference on Object-Oriented Programming, ECOOP '11*, LNCS. Springer-Verlag, July 2011.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of Javascript programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, June 2010.

[19] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. Programming Languages and Systems, 14th European Symposium on Programming, ESOP '05*, April 2005.

[20] P. Thiemann. A type safe DOM API. In *Proc. 10th International Workshop on Database Programming Languages, DBPL '05*, volume 3774 of *LNCS*. Springer-Verlag, 2005.